

信息物理融合系统 (CPS) 设计、建模与仿真 基于Ptolemy II平台

[美] 爱德华·阿什福德·李 (Edward Ashford Lee) 等编著
吴迪 李仁发 译

System Design, Modeling, and Simulation
Using Ptolemy II

System Design, Modeling, and Simulation

Using Ptolemy II



Claudius Ptolemaeus, Editor



机械工业出版社
China Machine Press

计 算 机 学 丛 书

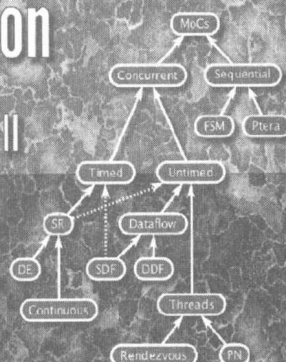
信息物理融合系统 (CPS) 设计、建模与仿真 基于Ptolemy II平台

[美] 爱德华·阿什福德·李 (Edward Ashford Lee) 等编著
吴迪 李仁发 译

System Design, Modeling, and Simulation
Using Ptolemy II

System Design, Modeling, and Simulation

Using Ptolemy II



Claudius Ptolemaeus, Editor



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

信息物理融合系统 (CPS) 设计、建模与仿真——基于 Ptolemy II 平台 / (美) 爱德华·阿什福德·李 (Edward Ashford Lee) 等编著 ; 吴迪, 李仁发译 . —北京 : 机械工业出版社, 2017.1

(计算机科学丛书)

书名原文 : System Design, Modeling, and Simulation: Using Ptolemy II

ISBN 978-7-111-55843-9

I. 信… II. ①爱… ②吴… ③李… III. 异构网络—研究 IV. TP393.02

中国版本图书馆 CIP 数据核字 (2017) 第 012647 号

本书版权登记号 : 图字 : 01-2015-7582

Authorized translation from the English language edition entitled System Design, Modeling, and Simulation using Ptolemy II (ISBN 978-1-304-42106-7) by Edward Ashford Lee, et al., Copyright © 2013 by Edward Ashford Lee.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanic, including photocopying, recording, or by any information storage retrieval system, without permission of by Edward Ashford Lee.

Chinese simplified language edition published by China Machine Press.

Copyright © 2017 by China Machine Press.

本书简体中文版由原书作者 Edward Ashford Lee 授权机械工业出版社独家出版发行。未经许可之出口, 视为违反著作权法, 将受到法律制裁。

本书是一本系统论述 CPS (集成了计算、网络和物理过程的信息物理融合系统) 建模问题的专著, 以 Ptolemy II 平台为基础, 广泛讨论了分层、异构系统的设计、建模和仿真技术。本书共分为三部分: 第一部分包括第 1 ~ 2 章, 主要介绍系统的设计、建模与仿真的基础概念; 第二部分包括第 3 ~ 11 章, 涵盖系统设计、建模和仿真中常用的计算模型, 其中每章都包括一个或一小类相关的计算模型, 并解释它们如何工作、怎样使用它们建立模型以及哪些种类的模型与计算模型可以更好地匹配; 第三部分包括第 12 ~ 17 章, 重点介绍由 Ptolemy II 提供的模型系统的内部组件, 讨论 Ptolemy II 跨模型计算的能力, 对于那些想要扩展 Ptolemy II 或者想用 Java 写自己的角色的读者来说, 可以从中得到具体指导。本书最后列出了大量的参考书目。

本书适合作为高等院校相关专业“嵌入式系统”课程的教材或教学参考书, 也可作为专业技术人员 在 CPS 系统建模过程中的参考书。

出版发行 : 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码 100037)

责任编辑 : 盛思源

责任校对 : 董纪丽

印 刷 : 北京诚信伟业印刷有限公司

版 次 : 2017 年 2 月第 1 版第 1 次印刷

开 本 : 185mm × 260mm 1/16

印 张 : 24.25

书 号 : ISBN 978-7-111-55843-9

定 价 : 79.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线 : (010) 88378991 88361066

投稿热线 : (010) 88379604

购书热线 : (010) 68326294 88379649 68995259

读者信箱 : hzsj@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问 : 北京大成律师事务所 韩光 / 邹晓东

文艺复兴以来，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域中取得了垄断性的优势；也正是这样的优势，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅筹划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自1998年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与Pearson, McGraw-Hill, Elsevier, MIT, John Wiley & Sons, Cengage等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出Andrew S. Tanenbaum, Bjarne Stroustrup, Brian W. Kernighan, Dennis Ritchie, Jim Gray, Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, Abraham Silberschatz, William Stallings, Donald E. Knuth, John L. Hennessy, Larry L. Peterson等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力相助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专门为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方式如下：

华章网站：www.hzbook.com

电子邮件：hzsj@hzbook.com

联系电话：(010) 88379604

联系地址：北京市西城区百万庄南街1号

邮政编码：100037



华章科技图书出版中心

本书是 Ptolemy II 项目近 20 年成果经验的总结，也是项目组成员共同智慧的结晶。Ptolemy II 项目是由美国加州大学伯克利分校教授、嵌入式系统领域的著名学者、CPS (Cyber-Physical System) 研究的倡导者和引领者 Edward Ashford Lee 负责的。他长期专注于并发、实时嵌入式系统领域的研究，开发了集系统设计、建模及仿真于一体的 Ptolemy II 系统平台。

为了避免讨论过于抽象，本书以 Ptolemy II 为基础，广泛讨论了分层、异构系统的设计、建模和仿真技术。本书共分为三部分。第一部分包括第 1 ~ 2 章，主要介绍系统的设计、建模与仿真的基础概念。第 1 章首先概述了异构系统规范化建模的指导性原则，从较高的角度对第二部分详细描述的计算模型 (Model of Computation, MoC) 进行概述。第 2 章提供了一个通过图形用户界面 Vergil 使用 Ptolemy II 的操作指南，使读者可以在系统设计过程中熟练掌握开源的 Ptolemy II 来进行实验。第二部分包括第 3 ~ 11 章，涵盖了系统设计、建模和仿真中常用的计算模型。每章都包括一个或一小类相关的计算模型，并解释了它们如何工作、怎样使用它们建立模型以及哪些种类的模型与计算模型可以更好地匹配。第三部分包括第 12 ~ 17 章，重点介绍由 Ptolemy II 提供的模型系统的内部组件，讨论了 Ptolemy II 跨模型计算的能力。对于那些想要扩展 Ptolemy II 或者想用 Java 写自己的角色的读者来说，可以从第三部分得到具体指导。本书最后列出了大量的参考文献。另外，书中提到所有的方法、实例都可以在项目网站 (<http://ptolemy.org/systems>) 上下载源代码。

本书的最初译稿来源于李仁发教授组织的两届“高性能嵌入式计算”研讨班所用教材之一。李仁发教授组织了本书的翻译工作。参加研讨班的全体博士生、硕士生为本书的共同译者，他们是：吴武飞、杜家宜、白洋、谌雨晴、黄雪伦、王娜、胡游、周兰花、黄一智、李坤明、黄晶、屠晓涵、马萌、彭理、何少芳、宋金林、袁娜、邓湘军、周佳、李万里和杨竞。吴迪、吴武飞负责全书的校对。吴迪负责全书的最后校审工作。

不同语言之间的转换是一件困难的事情。看似很直白的一个词，虽然理解其词义，但要换一种语言表达时往往煞费苦心，有的名词还很难给出确切的中文译名。译者力求忠实地表达书中所介绍的技术，保持原作者的行文风格。在本书翻译过程中，发现原书存在少量描述性错误，通过邮件与原作者确认核实后，在译文中直接进行了修订。限于时间以及译者水平和经验的不足，译文中难免存在不当之处，恳请读者批评指正。

本书的翻译工作得到了原书作者 Edward Ashford Lee 教授本人及 Ptolemy 项目组其他成员的鼎力支持，同时还得到湖南大学嵌入式与网络计算湖南省重点实验室同仁及机械工业出版社许多人士的帮助。对此，译者深表感谢。

错误反馈

如果你在阅读过程中发现本书中的错误或印刷错误，或者有任何改进的建议，请发送电子邮件到译者邮箱：enchnu@126.com。或者发送到原书作者邮箱：authors@leeseshia.org。

在邮件中，请注明该书的版本号和相关页面，非常感谢！

“我”上次发表著作是在一千九百年前^①。“我”很高兴从退休中复出，对以本人名字命名的工程（Ptolemy 工程）发表自己的看法。与“我”以往在天文和地理方面的工作相似，该项目也是对复杂系统进行处理。值得一提的是，类似“我”之前的许多著作，本书同样凝结了许多人共同的智慧和努力。

“我”以前在《The Almagest》(天文大全)中研究行星、太阳、地球和月亮的运动规律，这些运动都是并发交互过程（concurrent interacting process）。并且这些运动都是确定性的（deterministic），并不以神的意志为转移。这些模型的关注点不仅仅是对所观察到的行为进行精确匹配，更重要的是对行为的预测。类似地，Ptolemy 项目研究并发交互过程，并重点关注确定性模型。

理想情况下，求知欲推动着人类从迷信和盲目的信仰发展到逻辑和计量。现在所谓的“科学”深深根植于科学方法（scientific method），特别是在自然系统的研究中。利用科学方法，从设想开始，设计实验，并基于实验来对之前的设想下定论。当然，为了能够进行计量，待测量的工件或过程必须以某种形式存在。在“我”早期的研究中，不存在该问题，因为太阳、地球、月亮和行星是已经存在的事物。然而工程学科所关注的是人为的工件和过程，研究的是自然界中本不存在的系统。即便如此，科学方法也可用于并已经应用于工程设计中。工程师构建仿真和原型系统，将设想公式化，然后通过实验来进行设想的测试。

因为针对的是本不存在的工件和过程，所以工程设计不能单单基于科学方法。实验的目的是提高对所设计的工件、过程的认知。但是在进行实验前，必须将这些工件或者过程创造出来。在认识某些事物之前，不得不先把它们创造出来，这点注定了我们的设计会根植于“迷信”和盲目的信仰。

模型构造是与科学方法互补的重要科学部分。模型是物理现实的一种抽象，并且模型提供内视和行为预测的能力可以形成设想的核心思想，该思想核心等待被实验证实或证伪。建模本身更应归于工程学科，而非自然科学。从根本上讲，它并不是对于自然界已存在系统的研究。相反，它是人类主导的、对于自然界本身不存在事物的建造过程。一个模型本身就是一项工程。

好的模型甚至可以减少对计量的需求，因此可以减少对科学方法的依赖。比如，我们一旦有一个行星运动模型，我们就可以精确预测它们的位置，这样就减少了对它们位置测量的需要。计量的角色从确定行星位置转变为改善它们的运动模型以及检测模型对运动的影响（工程上称为“故障检测”（fault detection））。

无论在自然科学还是在工程中，模型都可以通过迭代方法来进行优化。“我”提出的以地球为中心的宇宙模型需要很多次迭代来修正，以逼近实验观测到的行星运动情况。最终模型的预测能力让“我”引以为豪。并且，基于这些模型的预测方法可以通过星盘机械化，这点同样让“我”感到自豪。即便这样，不得不承认，令人尊敬的同行哥白尼（Nicolaus

① 此处作者把自己比喻为古希腊天文学家、地心说体系创立者 Claudius Ptolemy。——译者注

Copernicus) 为行星运动提出了一个更好的模型(日心学说)。他的模型从概念上讲是更简单的。这是一种概念上的飞跃:我们可观测到的宇宙的中心,即我们所在的大地,并非一定是宇宙模型的中心。更进一步说,相对于物理世界,对于模型我们有更大的自由度,因为模型不需要被自然界所限制。即便如此,“我”所建立的模型在将近 1400 年的时间里也是一流的。

Ptolemy 项目确实是一项关注系统模型的研究。但是,该系统与“我”之前关注的系统有很大的不同。之前的那些系统都是自然界提供的,但是本书中的系统都是人造的。在本书中,建模的目的是优化系统,我们不可能对自然界给予的行星系统做任何优化。

简而言之,在与科学相反的工程中,模型要在被建模系统的设计阶段发挥作用。与科学一样,工程中的模型是可以被优化的,但是与科学不同的是,工程中的系统还可以被模型化。

更有趣的是,与科学不同的是,在工程中模型的选择对被建模的系统是有影响的。给予相同的目标,两位工程师可能会得出截然不同的系统设计和实现方案,这仅仅是因为他们开始阶段使用了截然不同的系统模型。进一步说,若两位工程师提出了不同的模型,其原因可能仅仅是他们在开始阶段使用了不同的工具来构建模型。一位用纸和笔建模的工程师与一位用软件工具建模的工程师得出的模型可能很不一样。结果就是,他们很可能得出迥异的系统设计。

针对复杂系统,本书收集了非常丰富的建模工具和技术。它们中的一些毫无疑问在以后会被优化,正如“我”自己提出的本轮(epicycle)模型,其建模的复杂性被哥白尼学派证明为不必要的。即使如此,本书的目的是向工程师提供目前可用的最好的建模技术。可以确信的是,我们将做得更好。

如何使用本书

本书是为需要对各种系统建模的工程师和科学家,以及想了解如何为复杂、异构系统建模的人而编写的。这些系统包括机械系统、电气系统、控制系统、生物系统等,更有趣的是,还包括结合了这些领域或者其他领域元素的异构系统。本书假设读者熟悉仿真和建模工具及其技术,但不要求对这些内容有深厚的背景知识。

本书重点强调 Ptolemy II 中已实现的建模技术。Ptolemy II 是一个开源的仿真和建模工具,用于对系统设计技术进行实验,尤其是那些涉及各种不同模型组合的系统。它是由 UC Berkeley 的研究人员开发的,并且由于过去 20 年里世界各地研究者的努力,它逐渐演变成一个复杂而精巧的工具。本书基于 Ptolemy II,对分层、异构系统的系统设计、建模和仿真技术进行了广泛的讨论。同时本书使用 Ptolemy II 来避免这些讨论过于抽象化和理论化。所有这些技术都由精心设计且测试效果良好的软件实现来支持。关于 Ptolemy II 更详细的底层软件架构以及更为细节的操作和基础理论,可以在知识点、参考文献和网络链接中找到。

本书共分 3 个部分。第一部分是“入门”。第 1 章概述了本书所涵盖的建模方式所蕴含的准则,并简要概述了多种计算模型(Model of Computation, MoC)。第 2 章介绍了怎样通过图形编辑器 Vergil 使用 Ptolemy II。对于那些想直接开始建模的读者,该章是个很好的起点。

第二部分包括第 3 ~ 11 章,涵盖了几乎所有的计算模型。每一章都包括一个或者一小

类相关的计算模型，并解释了它们怎样工作、怎样使用它们建立模型以及哪些种类模型与计算模型可以比较好地匹配。

第三部分讨论了 Ptolemy II 计算模型的可扩展性。对于那些想要扩展 Ptolemy II 或者想用 Java 写自己的角色 (actor) 的读者来说，第 12 章或许是最重要的一章，它描述了 Ptolemy II 软件架构。Ptolemy 是开源软件，并有完善的代码文档可供阅读。对于想要阅读代码并在此基础上做些工作的读者来说，该章可以提供很好的指引。第 13 章描述了用于规格化模型参数值和向角色 (actor) 中添加自定义函数的表达式语言。第 17 章描述了 Ptolemy II 标准库中包含的信号绘图仪 (signal plotter) 的功能。第 14 章讲解了 Ptolemy II 中的类型系统 (type system)。Ptolemy II 是一个复杂的类型系统，当提供一个强调类型系统来使安全最大化时，其设计旨在把建模工具的负担最小化 (通过强调类型推断而不是类型声明)。第 15 章讲述本体 (ontology)。本体能将单元部件、尺寸和概念与模型中的数值相关联，它增强了类型系统。同样，重点在于推断和安全。最后，第 16 章描述了 Ptolemy II 中的 Web 界面。具体地说，它解释了从模型中导出页面以及在模型中建立 Web 服务和服务器的功能。

致谢

本书在 Ptolemy 项目中描述的建模技术，经过了 UC Berkeley 项目成员的多年开发。根源可追溯到 20 世纪 80 年代。雏形来自 Messerschmitt (1984) 创建的名为 BloSim (Block Simulator) 的软件框架，用于仿真信号处理系统。Messerschmitt 的博士生 Edward A. Lee，受 BloSim 的鼓舞，开发了同步数据流 (Synchronous DataFlow, SDF) 计算模型 (Lee, 1986; Lee and Messerschmitt, 1987b) 和针对该模型的调度方法 (Lee and Messerschmitt, 1987a)。Lee 和他的学生接着完善了一个基于 Lisp 的软件工具 Gabriel (Lee et al., 1989)，通过它开发和完善了 SDF 计算模型。在 20 世纪 90 年代初期，Lee 和 Messerschmitt 开发的面向对象的方框图框架开始称为 Ptolemy (Buck et al., 1994) (现在称为 Ptolemy Classic)。在 20 世纪 90 年代后期，Lee 和他的小组基于最新的编程语言 Java (Eker et al., 2003)，开始了一个称为 Ptolemy II 的全新设计。Ptolemy II 发展了面向角色设计 (Lee et al., 2003) 和分层异构的主要思想。

软件的发展趋势在许多方面都反映了当前计算的演变。BloSim 用 C 语言编写。Gabriel 用 Lisp 语言编写。第一代 Ptolemy 系统，现在称为 Ptolemy Classic，用 C++ 编写。下一个版本 Ptolemy II，用 Java 编写。每一次变化都反映了人们利用最有效的技术解决实际设计问题的努力。Lisp 超过了 C 语言是因为其鲁棒性以及对复杂逻辑设计的适应性。C++ 超过了 Lisp 是因为其能更好地发展 (当时) 面向对象设计的概念 (特别是继承和多态)。Java 超过了 C++ 是因为其很好地支持了多线程和用户接口的可移植性。也许最重要的是，选择定期更换语言是为了强制这个小组重新进行设计，并扩充知识学习。

本书很大程度建立在基于 Java 的 Ptolemy II 上。即便如此，大部分的荣誉都应归功于 Ptolemy Classic (Buck et al., 1994) 的设计者，尤其是 Joseph Buck、Soonhoi Ha、Edward A. Lee 和 David Messerschmitt 等。

参与人员

许多人都对本书和 Ptolemy II 软件做出了很大的贡献。除了每章的作者外，还有很多人对本书贡献很大，他们包括 Shuvra S. Bhattacharyya、David Broman、Adam Cataldo、

Chihhong Patrick Cheng、Daniel Lazaro Cuadrado、Johan Eker、Brian L. Evans、Teale Fristoe、Chamberlain Fong、Shanna-Shaye Forbes、Edwin E. Goei、Jeff Jensen、Bart Kienhuis、Rowland R. Johnson、Soonhoi Ha、Asawaree Kalavade、Phil Lapsley、BilungLee、Seungjun Lee、Isaac Liu、Eleftherios D. Matsikoudis、Praveen K. Murthy、Hiren Patel、José Luis Pino、Jan Reineke、Sonia Sachs、Farhana Sheikh、Sun-Inn Shih、Gilbert C. Sih、Sean Simmons、S. Sriram、Richard S. Stevens、Juergen Teich、Neil E. Turner、Jeffrey C. Tsay、Brian K. Vogel、Kennard D. White、Martin Wiggers、Michael C. Williamson、Michael Wirthlin、Zoltan Kemenczy、Ye (Rachel) Zhou 和 Jia Zou 等。Christopher Brooks 是该软件的总负责人，因此软件拥有如此高的质量，他功不可没。其他参与者详见 <http://ptolemy.org/people>。

特别感谢 Jennifer White 提供全面的编辑帮助，他帮助提高了全书的编写质量。为本书的编辑提供帮助的人还有 Yishai Feldman、William Lucas、Aviral Shrivastava、Ben Zhang 和 Michael Zimmer 等。

出版者的话

译者序

前言

第一部分 入门

第 1 章 异构建模	2
1.1 语法、语义、语用	3
1.2 域和计算模型	4
1.3 模型在设计中的作用	5
1.4 角色模型	6
1.5 层次结构模型	7
1.6 异构建模的方法	7
1.7 时间模型	11
1.7.1 层次化时间	12
1.7.2 超密时间	12
1.7.3 时间的数字表示	14
1.8 域和指示器概述	15
1.9 案例研究	18
1.10 小结	22
第 2 章 图形化建模	23
2.1 开始	23
2.1.1 信号处理模型执行范例	24
2.1.2 模型的创建和运行	26
2.1.3 建立连接	28
2.2 令牌和数据类型	31
2.3 层次结构和复合角色	35
2.3.1 复合角色端口添加	36
2.3.2 端口类型设置	37
2.3.3 多端口、总线和层次结构	38
2.4 注释及参数设置	39
2.4.1 层次化模型中的参数	39
2.4.2 修饰元素	40
2.4.3 创建自定义图标	41
2.5 如何操作大模型	42

2.6 类和继承	43
2.6.1 实例中参数值的重写	45
2.6.2 子类和继承	45
2.6.3 模型间类的共享	47
2.7 高阶组件	49
2.7.1 MultiInstanceComposite 角色	49
2.7.2 IterateOverArray 角色	50
2.7.3 生命周期管理角色	52
2.8 小结	53

第二部分 计算模型

第 3 章 数据流	56
3.1 同步数据流	56
3.1.1 平衡方程	57
3.1.2 反馈回路	62
3.1.3 数据流模型中的时间	63
3.2 动态数据流	68
3.2.1 点火规则	68
3.2.2 DDF 中的迭代	71
3.2.3 将 DDF 与其他域结合	74
3.3 小结	77
练习	78
第 4 章 进程网络和会话	80
4.1 Kahn 进程网络	80
4.1.1 并发点火	83
4.1.2 PN 模型的执行停止	87
4.2 会话	88
4.2.1 多路会话	89
4.2.2 条件会话	90
4.2.3 资源管理	91
4.3 小结	92
练习	92
第 5 章 同步响应模型	96
5.1 固定点语义	97
5.2 SR 实例	98

5.2.1 非循环模型	98	7.5.1 状态机和 DE	161
5.2.2 反馈	99	7.5.2 数据流和 DE 组合	162
5.2.3 因果循环	106	7.6 无线和传感器网络系统	162
5.2.4 多时钟模型	106	7.7 小结	164
5.3 寻找定点	107	练习	164
5.4 定点逻辑	109	第 8 章 模态模型	166
5.5 小结	112	8.1 模态模型的结构	166
练习	112	8.2 转移	170
第 6 章 有限状态机	113	8.2.1 复位转移	170
6.1 Ptolemy 中的 FSM 创建	113	8.2.2 抢占式转移	171
6.2 FSM 的结构与执行	116	8.2.3 差错转移	172
6.2.1 转移条件定义	119	8.2.4 终止转移	174
6.2.2 输出动作	120	8.3 模态模型的执行	175
6.2.3 赋值动作和扩展有限状态机	120	8.4 模态模型和域	176
6.2.4 终止状态	122	8.4.1 数据流和模态模型	176
6.2.5 默认转移	123	8.4.2 同步响应和模态模型	181
6.2.6 非确定性状态机	124	8.4.3 进程网络和会话	181
6.2.7 立即转移	126	8.5 模态模型中的时间	181
6.3 分层 FSM	128	8.5.1 模态模型中的时间延迟	184
6.3.1 状态细化	129	8.5.2 本地时间和环境时间	185
6.3.2 分层 FSM 的优点	130	8.5.3 模式细化中的开始时间	187
6.3.3 抢占式转移与历史转移	130	8.6 小结	188
6.3.4 终止转移	132	练习	188
6.3.5 模态模型的执行模式	133	第 9 章 连续时间模型	189
6.4 状态机的并发复合	135	9.1 常微分方程	189
6.5 小结	137	9.1.1 积分器	189
练习	138	9.1.2 传递函数	191
第 7 章 离散事件模型	141	9.1.3 求解器	192
7.1 DE 域中的时间模型	142	9.2 离散和连续的混合系统	197
7.1.1 模型时间与实际时间	142	9.2.1 分段连续信号	197
7.1.2 并发事件	143	9.2.2 连续域中的离散事件信号	199
7.1.3 同步事件	144	9.2.3 离散时间的积分器重置	200
7.2 排队系统	149	9.2.4 狄拉克 δ 函数	201
7.3 调度	152	9.2.5 与 DE 互操作	204
7.3.1 优先级	154	9.2.6 定点语义	205
7.3.2 反馈回路	155	9.3 混合系统和模态模型	206
7.3.3 多线程执行	157	9.3.1 混合系统和不连续信号	208
7.3.4 调度局限性	159	9.4 小结	210
7.4 芝诺 (Zeno) 模型	160	练习	210
7.5 其他计算模型与 DE 的组合	161		

第 10 章 计时系统建模	211
10.1 时钟	211
10.2 时钟同步	214
10.3 通信延时建模	217
10.3.1 固定和独立的通信延时	217
10.3.2 共享资源竞争行为建模	219
10.3.3 复合切面	222
10.4 执行时间建模	223
10.5 分布式实时系统的 Ptidcs 模型	225
10.5.1 Ptidcs 模型的结构	226
10.5.2 Ptidcs 组件	231
10.6 小结	233
第 11 章 Ptera: 面向事件的 计算模型	234
11.1 扁平模型的语法和语义	234
11.1.1 入门实例	235
11.1.2 事件参数	236
11.1.3 取消关系	237
11.1.4 同时事件	237
11.1.5 潜在的非确定性	237
11.1.6 LIFO 和 FIFO 策略	238
11.1.7 优先级	239
11.1.8 事件命名及调度关系	239
11.1.9 原子性设计	239
11.1.10 面向应用的实例	240
11.2 层次模型	242
11.3 异构组合	243
11.3.1 Ptera 与 DE 组合	243
11.3.2 Ptera 与有限状态机组组合	245
11.4 小结	246

第三部分 建模的基础结构

第 12 章 软件体系结构	248
12.1 包结构	248
12.2 模型结构	249
12.3 角色语义和计算模型	253
12.3.1 执行控制	253
12.3.2 通信	256
12.3.3 时间	257

12.4 在 Java 中设计角色	258
12.4.1 端口	261
12.4.2 参数	262
12.4.3 端口和参数耦合	263
12.5 小结	264
第 13 章 表达式	265
13.1 简单算术表达式	265
13.1.1 常量与直接值	265
13.1.2 变量	267
13.1.3 运算符	268
13.1.4 注释	269
13.2 表达式的应用	269
13.2.1 参数	270
13.2.2 端口参数	270
13.2.3 字符串参数	271
13.2.4 表达式角色	272
13.2.5 状态机	272
13.3 复合数据类型	273
13.3.1 数组	273
13.3.2 矩阵	275
13.3.3 记录	276
13.3.4 联合体	278
13.4 令牌运算	279
13.4.1 调用方法	279
13.4.2 访问模型元素	279
13.4.3 类型分配	280
13.4.4 函数定义	281
13.4.5 高阶函数	281
13.4.6 模型中的函数调用	282
13.4.7 递归函数	283
13.4.8 内置函数	284
13.5 空值令牌	287
13.6 定点数	287
13.7 单位	288
13.8 函数表	290
第 14 章 类型系统	298
14.1 类型推断、转换和冲突	298
14.1.1 自动类型转换	300
14.1.2 类型约束	302
14.1.3 类型声明	303

14.1.4 反向类型推断	304	15.3.3 维度之间的转换	327
14.2 结构化类型	305	15.4 小结	329
14.2.1 数组	305	第 16 章 Web 接口	330
14.2.2 记录	306	16.1 导出到网络	330
14.2.3 联合体	307	16.2 Web 服务	341
14.2.4 函数	307	16.2.1 Web 服务器的架构	341
14.3 角色定义中的类型约束	307	16.2.2 构建 Web 服务	343
14.4 小结	312	16.2.3 使用 cookie 在 客户端存储数据	346
第 15 章 本体	314	16.3 小结	351
15.1 创建和使用本体	315	练习	351
15.1.1 本体创建	316	第 17 章 信号显示	352
15.1.2 约束创建	318	17.1 可用绘图仪概述	353
15.1.3 抽象解释	321	17.2 绘图仪定制	355
15.2 错误查找和最小化	322	参考文献	358
15.3 单位系统创建	326		
15.3.1 什么是单位	326		
15.3.2 基本维度和推导维度	327		

第一部分

System Design, Modeling, and Simulation using Ptolemy II

人 门

本书第一部分主要介绍系统的设计、建模与仿真。第1章首先概述了异构系统规范化建模的指导性原则，从较高的角度对将在第二部分中详细描述的计算模型（Model of Computation, MoC）进行概述。另外，第1章还提供了一个高度简化的研究案例（一个发电机组），该案例阐明了多种不同计算模型在复杂系统设计中所起的作用。

第2章提供了一个利用图形用户界面 Vergil 使用 Ptolemy II 的操作指南。本书目标之一就是使得读者能够在系统设计过程中利用开源的 Ptolemy II 进行实验。这章的目的在于提供足够的信息，以使读者成为 Ptolemy II 的合格使用者。关于如何对 Ptolemy II 进行扩展，读者可参考本书第三部分。

异构建模

Edward A. Lee

当前的许多工程系统通常都结合了异构且复杂的子系统。例如一辆汽车，就可能结合了一个复杂的发动机、很多的电子控制单元（Electronic Control Unit, ECU）、引擎控制系统、车身电子控制系统（用于控制车窗和门锁）、娱乐系统、空调控制和通风系统，以及各种安全子系统（如安全气囊）。每个子系统可能又由软件、电子及机械部分联合组成。实现如此复杂的系统的确是一项挑战，尤其在于：即使最小的子系统也跨越了多个工程学科领域。

这些复杂系统同样也对设计工具带来了挑战。工程师通过使用设计工具对系统进行规格化、设计、仿真及分析。如今，仅仅是画出机械结构的草图，然后列出一些等式描述机械部分之间的交互是不够的。而且，无论是完全依靠机械部分的 3D 建模软件工具，还是依靠用于软件系统的基于模型的设计工具，都是不够的。各个领域（机械、软件、电子、通信网络、化学、流体动力学以及人为因素）之间相互联系的复杂性削弱了只适用于单个领域工具的有效性。

本书重点针对信息物理融合系统（Cyber-Physical System, CPS）（Lee, 2008a, 2010a ; Lee and Seshia, 2011），这种系统将物理动力学与计算和网络结合。CPS 需要通过模型组合的方式将物理过程的连续动态（通常用微分方程描述）与软件模型集成在一起。有些应用需要结合组件间的计时交互和传统算法计算[⊖]来实现，这种情况下混合模型是最有效的。在那些算法组件间有并发[⊕]交互（concurrent interaction）的传统软件系统中，也可以使用混合模型。

补充阅读：关于术语“CPS”

术语“CPS”（Cyber-Physical System）出现于 2006 年前后，是由美国国家科学基金会的 Helen Gill 所提出。对于 William Gibson 的“赛博空间”（cyberspace）一词，我们都很熟悉。Gibson 在小说《Neuromancer》（神经漫游者）中用这个词来表示用于人类之间相互通信的计算机网络媒介。我们试图将术语赛博空间与 CPS 联系起来，但是术语 CPS 的根源更深远。或许，把 cyberspace 和 CPS 视为由同一词语“控制论”（cybernetics）演

⊖ 算法指的是对解决某个问题所需的有限步骤序列描述。物理过程很少呈现如步骤序列那样的结构；相反，它们的结构组织类似于并发组件（concurrent component）间的连续交互。

⊕ 并发，由拉丁文中“concurrere”一词而来，意味着同时发生；它在计算机科学中是指两个或多个步骤序列的任意交错。但是，这只是对基本概念的专业解释。本书认为“并发”这个概念是指同步操作，而并不表示交错或者一个步骤序列。特别地，两个连续过程可以并发操作，而并不要求它们被直接表示成一系列步骤。比如，放入一桶水中的一个电阻加热元器件。增大通过这个加热元器件的电流，会使水温上升。电流与水温都是连续过程，并且这两个过程相互影响。但是，这两种过程中的任一种都无法合理地表示为一系列步骤，并且这个总体过程也不是这些系列步骤的交错。

变出的同源词会更准确一些。

术语“控制论”由美国数学家 Norbert Wiener (Wiener, 1948) 提出, 他对控制系统理论的发展有巨大影响。在第二次世界大战期间, Wiener 发明了高射炮的自动瞄准和射击技术。虽然他使用的机制并不涉及数字计算机, 但涉及的原理与现今很多基于计算机的反馈控制系统相似。Wiener 发明的这个词起源于希腊语 κυβερνητης (kybernetes), 意思是舵手、长官、领航员或船舵。该比喻对控制系统来说是很恰当的。

Wiener 将他对“控制论”的理解描述为控制和通信的结合。他对控制的概念根植于闭环反馈 (closed-loop feedback), 这里的控制逻辑由对物理过程的计量结果驱动, 然后控制逻辑反过来也驱动物理过程。即使 Wiener 没有使用数字计算机, 但控制逻辑从效果上来讲就是一种计算, 因此“控制论”就是物理过程、计算和通信三者的结合。

Wiener 当时没有料到数字计算和网络的巨大影响力。“Cyber-Physical System”可能被模糊地解释为“赛博空间”和物理过程的结合, 这一点削弱了 CPS 可能具有的巨大影响力。CPS 对信息技术带来的显著影响, 甚至超越了 Wiener 时代最疯狂的想象。

1.1 语法、语义、语用

在撰写本书的过程中, 工程工具和技术正处在巨大变化时期。这种变化推动着我们的工作, 使我们有能力适应系统日益增加的复杂性和异构性。过去, 整个行业都围绕着为单一的工程领域提供设计工具, 比如数字电路、软件、3D 机械设计、采暖及通风领域。如今, 我们看到越来越多设计工具的整合和组合; 独立工具常常扩展到工具套件, 并提供传统领域以外的功能。这种变革也带来了一些问题, 即不良的集成会导致一些不可预期的行为。工具集成常常导致怪异集成 (frankenware)^①, 也就是说, 由几乎不兼容的工具组成的不稳定组合很难得到有效维护和使用。

另外, 在一个相对狭窄的领域中一贯良好的工具, 在更广泛的领域中却并非那么有效。今天复杂的设计工具涉及语法 (syntax) (如何表示一个设计)、语义 (semantics) (一个设计表示什么, 以及它是如何工作的)、语用 (pragmatics) (Fuhrmann and von Hanxleden, 2008) (工程师应该如何使设计可视化, 并对其进行编辑和分析) 的复杂组合, 如图 1-1 所示。当一个设计工具被用于它的原始使用领域之外时, 或者用于和其他工具的组合之中时, 不兼容的语法、未充分理解的语义以及不一致的人机界面都有可能使得其不能有效使用。

会出现语法上的不兼容, 是因为不同设计结构的本质是不同的 (比如, 软件的语法和 3D 设计几乎没有共同之处)。但是出现语法不兼容的一个更普遍的原因是, 各种工具是在不同的工程领域以不同的技术开发的。以工具的语用为例, 对于如何管理设计文件, 如何追踪改动也会因不同发生时刻而不同。语义的差异也有一定偶然性, 不同的理解会加大这种差异。语用在不同领域中可能具有不同意思。比如说, 在控制工程师和软件工程师眼中, 同一个框图可能代表着完全不同的意思。

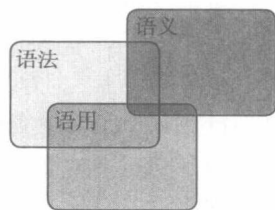


图 1-1 如今的设计工具涉及语法、语义和语用的复杂组合

① 术语“frankenware”源于 Christopher Brooks。

本书使用 Ptolemy II 对异构建模的几个关键概念进行了检验。Ptolemy II^①是一个开源的建模和仿真工具。与大多数其他设计工具不同，Ptolemy II 从开发之始就专注于异构系统。Ptolemy 项目（UC Berkeley 正在进行的一项研究）的一个关键目标就是将不同领域之间语法、语义和语用之间的差异最小化，并将不同领域设计之间的互操作性最大化。因此，Ptolemy II 为 CPS 系统的设计提供了一个有数的实验环境。

Ptolemy II 集成 4 种不同类型的语法：框图、弧线图（bubble-and-arc）图、命令式程序和算术表达式。这些语法是互补的，这使得 Ptolemy II 能够处理各种设计领域的问题。框图用来表示相互通信的组件之间的并发关系；图用来表示状态或模式的顺序；命令式程序用来表示算法；算术表达式用来表示函数的数值计算。

Ptolemy II 也集成了一些语义域。尤其是对于框图来说，它的语义有多种可能，彼此都有明显的不同。框图之间的连接表示设计中组件之间的交互（interaction）。但是什么类型的交互呢？是异步通信（如寄信）？是会话形式的通信（如打电话）？还是数据的定时更新（如在同步数字电路中那样）？在交互中，时间是否起到了作用？交互是离散的还是连续的？为了支持异构建模，Ptolemy II 支持以上提到的所有需求，并且它还可以被扩展以支持更多的需求。

1.2 域和计算模型

Ptolemy II 中的语义域（semantic domain），通常称为域（domain），它定义了设计中两个组件交互的“物理定律”。它为组件之间的并发执行以及两个组件之间的通信（如前文所述）提供了管理规则。这种规则的集合称为**计算模型**（Model of Computation, MoC）。在本书中，从技术上看尽管域是计算模型的实现，但术语“计算模型”和“域”是可替换的。计算模型是一个抽象模型，然而域是模型在软件上的具体实现。

模型规则分为三类。第一类规则指定了组件的构成要素，在本书中，一个组件一般是一个角色（actor），在下文中将给予更精确地定义。第二类规则指定执行和并发机制：角色调用是按序的？同时的？还是非确定性的？第三类规则指定通信机制：角色之间怎样交换数据？

本书中讨论的每一个计算模型都有很多可能的变体，这些变体中很多已经在其他的建模工具中实现了。本书把重点放在 Ptolemy II 中实现的计算模型，以及那些具有易读且书写良好的语义模型上^②。为了进一步阐述，我们也提供了其他一些有用的、还未在 Ptolemy II 中实现但已在其他工具中实现的计算模型的简要说明和索引。

为了支持异构系统的设计，Ptolemy II 域之间可以交互操作。这要求语义域之间有一定程度的协议。但是，当不同的工具被分别独立设计再组合到一起时，这种协议几乎是不存在的。Ptolemy II 中域之间交互的法则在多篇论文中有所描述（Eker et al., 2003；Lee et al., 2003；Goderis et al., 2009；Lee, 2010b；Tripakis et al., 2013）。本书重点在于域的互操作性的实践环节，而不是理论。

使用统一的、一致的软件系统使我们可以专注于域的交互操作，而不必过多担心不同工具集成过程带来的不兼容性问题。比如说，Ptolemy II 的类型系统（type system）（它

① Ptolemy II 可从 <http://ptolemy.org> 下载。

② 在本书的电子版中，大多数模型插图的图注都提供了超链接，你可以在线浏览这些模型。若你的机器支持 Java，你还可以编辑这些模型并执行它们。

定义了可以被各种计算组件所使用的数据类型)被所有的域、状态机符号以及表达式语言(expression language)所共享。域有能力推测和验证数据类型是否恰当;这个功能可以在异构模型中的多个域之间无缝地工作。同样,语义中包含时间概念的域共享一个通用的时间表达方式以及一个(多样)时间模型。Ptolemy II 中的域均可使用相同的图形编辑器,且均使用 XML(可扩展标记语言架构)文件是存储设计。该协议消除了异构模型组合中存在的很多实际障碍。这允许我们集中精力关注异构集成带来的好处——最重要的是,即便设计是异构的,我们也能够选择与问题最匹配的域。

1.3 模型在设计中的作用

本书为在 Ptolemy II 中理解和建立模型提供了一个框架,更广泛地说,这是理解建模中的关键问题并对复杂异构系统进行仿真的一个框架。该主题如此广泛,使得单一的一本书难以涵盖所有对系统设计师有用的技术。在本书中,我们关注那些描述**动态系统(dynamics)**的模型,或者那些描述一个系统或子系统如何随时间变化的模型。我们并不关注那些主要侧重静态结构设计的技术(例如,软件模型或 3D 模型设计所用到的 UML 类图)。因此,本书中所有的模型都是可执行的。我们把模型的执行称为**仿真(simulation)**。

图 1-2 中展示了系统实现过程的三个主要部分:建模、设计和仿真。**建模(modeling)**是一个通过模拟而获得对系统更深入了解的过程。模型模拟了系统并反映系统的属性。动力学模型具体说明了系统需要做什么,即它对所处环境的变化做出怎样的反应,以及随时间发生怎样的演化。**设计(design)**是对工件(如软件组件)的结构化创建,用以实现特定的功能。它指定一个系统如何实现所需的功能。**仿真(simulation)**展示了模型在特定环境中的行为。仿真是进行设计分析的一种简单的形式,它的目标是使得我们可以观测设计的属性,并使设计可测试(testing)。在本书中,我们所讨论的模型可以经受更为详细复杂的分析形式,比如说形式验证(formal verification)。在其最一般的形式中,分析是一个可通过解剖或划分成更小、更容易分析的部件(piece)来获得对系统更深入了解的过程。它指明了为什么系统能做什么(或不能做什么)、模型应该做什么。这里,除了仿真,我们将其他的分析技术都留给其他的书籍来解释。

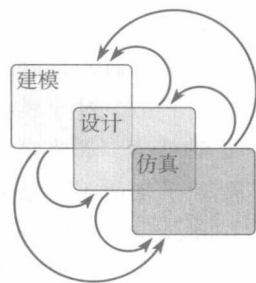


图 1-2 建模、设计、仿真的迭代过程

如图 1-2 所示,设计过程中的三个部分是相互重叠、相互循环的过程。通常,设计过程始于建模,目的是为了理解问题及制订解决问题的策略。

建模在现代设计过程中扮演核心角色。**基于模型的设计(model-based design)**的主要原则是:最大限度地利用建模去构建更好的设计。模型必须相当准确,当然,它们也必须是可理解和可分析的,这样才是有效模型。一个模型必须具有一个明确的含义(清晰的语义),才能做到可理解和可分析。

模型用**建模语言(modeling language)**来表达。比如,有些程序是用 Java 或 C 语言编写的,所以这些编程语言实际上就是程序的建模语言。如果表述模型的语言具有清晰的、明确的意义,那么这个建模语言就有**强语义(strong semantics)**。例如,以下例子所示,Java 的语义强于 C。

例 1.1 假设一个程序的参数是 int 型。在 Java 语言中,这个数据类型有良好的定义,但在 C 语言中就没有。例如,在 C 语言中,int 型可能表示 16 位的整型也可能表示 32 位的

整型。程序的行为可能完全不同，这取决于提供了哪种实现。特别是，16 位整型比 32 位整型更容易发生溢出。

补充阅读：系统模型与系统实现

模型必须被谨慎地使用。**Kopetz 准则**（以维也纳工业大学的 Hermann Kopetz 教授的名字命名）告诉我们：很多我们称为系统属性的性质（确定性、实时性、可靠性），实际上并不是已实现系统的属性，而是系统模型的属性。

Golomb (1971) 强调理解模型和被建模事物之间的区别。他有一个著名的说法“你永远不可能从地图上钻出石油来！”但是，这丝毫不能降低地图的价值！考虑**确定性** (determinism) 即可。如果模型为每个特定的输入都只产生一个独立定义的输出，则模型是**确定的** (determinate)。如果对于任意特定的输入有多个可能的输出，则模型是**非确定的** (nondeterminate)。即使这看起来是一个简单的定义，但是这其中也有许多微妙之处。“特定输入”是什么意思？输入到达的时间重要吗？“唯一确定的输出”是什么意思？需要考虑当硬件执行失败时系统会做出怎样的行为吗？

任何有关物理上“已实现的”系统的确定性陈述，从根本上来说，是一个宗教或哲学论断，是不科学的。几乎可以断言的是，没有真正的物理系统是确定的。例如，当它被毁坏时是如何表现的？或者反过来断言，物质世界中的一切都是注定的，显然这是一个很牵强的却又难以反驳的、无用的概念。

然而，对于模型却可以明确断言其确定性。例如，由一种编程语言定义的程序可能是明确的，因为它的返回值仅仅取决于它的参数。甚至可以说，没有执行的程序实际上也是确定的（可能因硬件失效而无法产生返回值）。程序是一个定义在**形式框架** (formal framework) (语言的语义) 下的**模型** (model)。它抽象地对机器的执行进行建模，省略了一些信息。对于这种模型来说在任何时间提供输入都无法区别，因此“时间”不属于“特定输入”的范围。输入/输出仅仅是数据，而程序定义了输入和输出之间的关系。Box 和 Draper (1987) 支持这种模型的观点，他们表示，“本质上，所有的模型都是错的，但是有些是有用的”。模型的有用性取决于模型的**保真度** (fidelity)，即模型模拟系统的准确程度。但模型永远只是一种逼近。

很多流行的建模语言都是基于**弱语义** (weak semantic) 的框图。很多建模语言常采用框图符号而未对框图之间的连线给出精确的定义，对组件间的交互描述不够清晰（参看 1.6 节补充阅读：UML、SysML 和 MARTE 中的角色）。弱语义的建模语言会更加难以分析。它们的价值反而在于利于人们在非形式化地交流设计概念时使用。

1.4 角色模型

Ptolemy II 基于一类**面向角色的模型** (actor-oriented model)，或简单称为**角色模型** (actor model)。**角色**是可以并发执行且可以通过端口彼此共享数据的一些组件。

例 1.2 考虑图 1-3 中的 Ptolemy 模型。这个模型有 3 个角色，每个角色有一个端口。角色 A 通过它的端口向角色 B 和 C 发送信息（标有“关系” (Relation) 的菱形表示 A 的输出流向 B 和 C）。

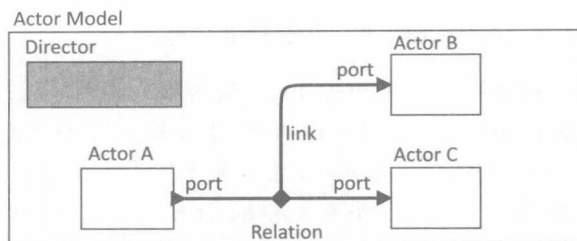


图 1-3 一个简单的角色模型的视觉演示

通过一个端口的所有信息的集合称为信号（signal）。例子中的指示器（Director）方框指定了域（从而也指定了计算模型）。本书的大部分工作就是解释在 Ptolemy II 中实现的各种域。

1.5 层次结构模型

复杂系统的模型通常很复杂。构建良好的复杂系统模型是一门艺术（模型工程（model engineering）艺术）。一个复杂系统的好模型提供了该系统相对简单的视角，以便于理解和分析。构建简单视角模型的关键方法是使用层次化结构建模（hierarchy），这样，对于在一个模型中看似单一组件的东西，从其内部看来又是一个模型。

一个层次化的角色模型如图 1-4 所示。它是图 1-3 的细化，它显示了角色 A 和 C 本身也是角色模型。一个原子角色（atomic actor）（原子来自希腊神话的 *atomos*，意味着不可分割的）是指其不能被定义为角色模型。相反，复合角色（composite actor）本身就是其他角色的组合。图中的端口 p 和端口 q 连接两个层级。例如，从 D 开始的一次通信，经过端口和上一层级，将到达角色 E。

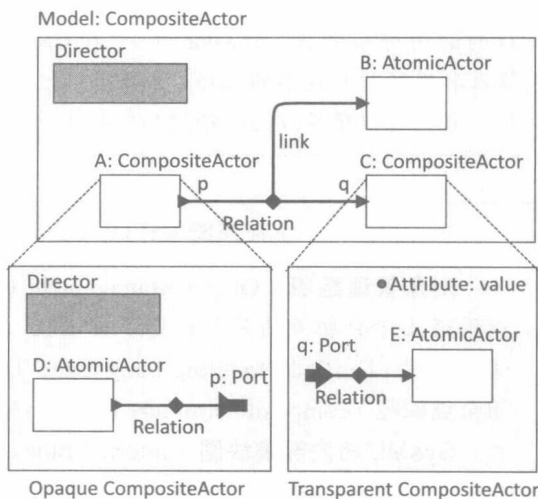


图 1-4 一个分层模型，由一个顶层的复合角色（composite actor）和两个子模型构成，每个子模型都是一个复合角色

1.6 异构建模的方法

异构建模有很多种方法（Book et al., 2008）。在 **多视图建模**（multiview modeling）中，为同一系统构建相互独立且不同的多个模型，用来从不同切面（aspect）对系统进行建模。比如说，一个模型描述的可能是动态行为，但另一个模型描述的可能是物理设计和封装。在 **无固定结构的异构**（amorphous heterogeneity）问题中，同一个模型中任意地组合了不同的建模风格，对结构并无益处。比如，一个模型中的一些组件交互可能要使用 **会话通信**（rendezvous messaging）（即一次通信发生之前发送者和接收者都要做好准备），但是另一些组件交互使用 **异步通信**（即接收者在发送者发送完之后的非确定时间内接收信息）。在 **层次化的多重建模**（hierarchical multimodeling）中，不同建模风格被层次化地组合，以便利用每一种风格独有的功能和表达能力。

补充阅读：关于术语“角色”

面向角色的建模 (actor-oriented modeling) 概念与“角色”这个术语密不可分。角色这个术语，在 20 世纪 70 年代由 Hewitt 描述自动推理代理 (autonomous reasoning agent) 时被引出 (Hewitt, 1977)。在 Agha 等人关于描述并发机制 (concurrency) 的形式化模型的工作中，该术语又发生了演变 (Agha et al., 1997)。Agha 的角色，每一个都具有独立的控制线程，且角色之间的通信使用异步通信。Dennis 的离散原子计算的数据流模型 (dataflow models of discrete atomic computations) (Dennis, 1974) 中，也使用了“角色”这个术语，该模型对有效输入的反应是将产生的输出作用到其他角色。

本书中，术语“角色”囊括了一大类并发模型。它们通常比通用的消息传递更具约束性，并且它们不必与数据流语义相一致。这里的“角色”从概念上讲仍是并发性的，但不同于 Agha 的“角色”，它们不需要具有自己的控制线程。不同于 Dennis 的“角色”，它们不需要由输入来触发。另外，虽然通信仍是由某种形式的消息传递实现，但其不必是异步的。

角色是系统中的组件，且它们可以类比为对象这种面向对象设计中的软件组件。在流行的面向对象语言 (如 Java、C++ 和 C#) 中，对象的接口主要是方法 (method)，也就是修改和观察对象状态的程序。相比之下，角色的接口主要是端口，它被用来接收和发送数据。它们与程序所做的顺序控制转移不相同，因此更适合并发模型。

补充阅读：UML、SysML 和 MARTE 中的角色

对象管理组织 (Object Management Group, OMG) 将角色模型中一些常见的、与框图语法十分相关的符号进行了标准化。本书中的角色模型与 UML 2 (第 2 版的统一建模语言 (Unified Modeling Language, UML)) (Bock, 2006; Booch et al., 1998) 的组合结构图 (composite structure diagram) 相关，或者说，与由它衍生的 SysML 直接相关。SysML 的内部模块图 (internal block diagram) 表示法，尤其是流端口 (flow port) 的使用，与角色模型密切相关。在 SysML 中，角色称为“模块”。(“角色”这个词在 UML 用于另一个目的)。

然而，SysML 强调如何表示模型图 (可视化表示)，但关于模型图的意义以及模型图如何操作的一些细节 (语义) 还悬而未决。比如说，虽然 SysML 宣称“流端口用于异步、广播或无反馈交互” (OMG, 2008a)，但 SysML 中没有计算模型。虽然不同的 SysML 工具可能赋予流端口不同的行为，但所有行为仍然符合标准。一个 SysML 模型可以表示多个设计，且模型的行为依赖于解释模型所用的工具。SysML 的重点是进行符号的规范，并不是规范符号的意义。

相比之下，Ptolemy II 的语义模型重点在模型的语义，而不是如何表示模型 (视觉或其他方面)。视觉符号是次要的，且实际上它并不是 Ptolemy II 模型的唯一表示方法。Ptolemy II 的指示器 (director) 赋予了模型一个很精确的意义。该具体的意义确保对于不同的观察者来说，模型所代表的意义都相同，并且使异构模型之间的相互操作成为可能。

与 SysML 相比，MARTE (实时和嵌入式系统建模与分析) 更加强调模型的行为 (OMG, 2008b)。它避免“约束”执行语义，使标准变得灵活，可以对很多流行的实时

建模技术进行表述。相比之下，Ptolemy II 的重点并不在于介绍现有的设计经验，而更多的是提供精确的、良好定义的系统行为模型。有趣的是，MARTE 包括了一个多样时间模型（multiform time model）（Andret et al., 2007），这与 Ptolemy II 中支持的模型是一样的。

软件工程师熟悉的一个例子是状态图（Statecharts）（Harel, 1987），它层次化地结合了同步并发组合（synchronous concurrent composition）与有限状态机。层次化建模的另一个例子就是协同仿真（cosimulation），两个不同的仿真工具通过标准化的接口结合在一起，就像 Simulink 中的 S 函数接口或者 Modelica 协会的功能模型接口（Functional Mockup Interface, FMI）^①。同样，也可能通过创建十分灵活的或者部分指定的建模框架来支持异构建模。这些框架可以被修改以便重写感兴趣的模型。这种方法的不利之处就是弱语义。Ptolemy II 的目标是实现强语义，但要包含异构性质并为异构模型提供并发交互机制。

如图 1-4 所示，一个复杂模型可以划分为由内嵌子模型组成的层次树。在每一层中，子模型可以联合在一起形成由相互作用的角色组成的网络。Ptolemy II 限制层次中的每一层都是同构的，使用一个通用的计算模型。之后，这些同构网络可以被层次化组合起来，形成一个更大的异构的模型。这种方法的好处就是，系统中的每一部分都可以使用与其过程需求最匹配的模式进行建模——尽管每一个计算模型都提供了强语义以确保它相对容易理解、分析和执行。

在 Ptolemy II 中，一个指示器规定了一个模型的语义。在图 1-4 中，有两个指示器。位于顶层的指示器规定了角色 A、B、C 之间的交互。因为 C 内部没有指示器，所以顶层的指示器管理着它和 E 的交互。角色 C 称为透明复合角色（transparent composite actor），它包含的模块对于它的指示器是可见的。

相比之下，角色 A 内部包含了另一个指示器。该指示器管理子模型之间的角色交互（在这个简单的例子中，只有一个这样的角色，但实际上可以有多个）。角色 A 称为不透明复合角色（opaque composite actor），它的内容对 A 的外部指示器来说是不可见的。为了区分这两个指示器，称外部的指示器为执行指示器（executive director）。对于执行指示器来说，角色 A 看起来是一个原子角色。但实际上 A 的内部包含了另一个模型。

在层次结构中，位于不同层的指示器不需要实现相同的计算模型。因此，使用不透明复合角色是 Ptolemy 实现层次化多重建模及协同仿真的方法。

补充阅读：模型的多元化

奥卡姆剃刀定律是科学界和工程界的一条准则，它鼓励选择那些对假设条件、基本条件或实体要求最低的理论 and 假说来解释一个给定的现象。该定律可表达为“实体的增加不可超出必要的范围”或者“若无需要，勿增实体”（Encyclopedia Britannica, 2010）。这条定律归功于 14 世纪的英国逻辑学家、神学家和方济会修士奥卡姆的威廉（William of Occam）。

即使这条定律的价值令人叹服，但其仍有局限。比如说，Immanuel Kant 想要减弱

^① <http://www.functional-mockup-interface.org>。

奥卡姆剃刀原则的影响，他声称“生命体的多样性不应被草率地减少”（Smith, 1929）。爱因斯坦据此评论，“凡事应力求简单，但不能过于简单”（Shapiro, 2006）。

当应用于设计技术时，奥卡姆剃刀定律使我们偏向于使用更少更简单的设计语言和符号。但是，经验表明冗余和多样性也可以是有利的。比如说，即使UML的类图要表达的信息已经编码在一个C++程序里了，但使用UML类图仍然是有益处的。使用UML的用例图也是有价值的，有此概念无法用C++程序编码，也不能（直接）用UML类图表示。这三种表示方法服务于不同的目的，虽然它们表示的是相同的底层进程。

许多不同的符号被用于UML和它的衍生物中，这个事实与奥卡姆剃刀定律背道而驰。具有讽刺意义的是，诞生于20世纪90年代的统一建模语言（UML）正是为了减少用于表示面向对象的软件架构符号的多样性（Booch et al., 1998）。那么，这种违背剃刀定律的做法有哪些好处呢？

软件系统的设计本质上是一种创造性的过程。工程师创建了本不存在的程序。奥卡姆剃刀定律应用于创造性过程时应当谨慎，因为当有多样的、可以达到理想效果的媒介时，创造力会很旺盛。通过提供比C++源码更抽象的符号，UML促进了创造性过程，并且这些符号有利于设计性的实验和设计思想的交流。

补充阅读：关于异构模型

有些作者使用术语**多范式建模**（multi-paradigm modeling）来描述混合了多种计算模型的方法（Mosterman and Vangheluwe, 2004）。Ptolemy II重点在于将角色和多范式建模相结合的技术。这种混合模型的一种早期解决方案在Ptolemy Classic（Buck et al., 1994）中实现，Ptolemy Classic是Ptolemy II的早期版本（Eker et al., 2003）。受到Ptolemy方法的影响，SystemC也能够实现多重计算模型（Patel and Shukla, 2004；Herrera and Villar, 2006）。ModHel'X（Hardebolle and Boulanger, 2007）和ForSyDe（Jantsch, 2003；Sander and Jantsch, 2004）也是如此。

另外，还有一种方法支持混合并发和通信机制，而不受层次结构的束缚（Goessler and Sangiovanni-Vincentelli, 2002；Basu et al., 2006）。其他一些研究者使用了创造性的方法来解决异构性问题（Burch et al., 2001；Feredj et al., 2009）。

使用**工具集成**（tool integration）也是可行的，即不同的建模工具通过转换语言或者协同仿真的方式结合起来（Liu et al., 1999；University of Pennsylvania MoBIES team, 2002；Gu et al., 2003；Karsai et al., 2005）。但使用这种方式是具有挑战性的，且受制于脆弱的工具链。关于工具如何被扩展、在哪个部分可以扩展以及如何进行工具之间的集成，许多工具也是缺乏相应文档的；实现和维护这样的集成需要极大的努力。面临的挑战包括：API的不兼容、不稳定或无正式文档的API、不清晰的语义、语法不兼容以及不可维护的代码库等问题。事实证明，工具集成要完成异构设计是非常艰巨的。相对于关注工具集成中的软件问题，一种更好的方法则是专注于语义的集成。只有在对语义交互的良好理解的基础上，才有可能出现良好的软件架构。

在Ptolemy中，每一个模型都包括一个指示器。指示器用来指定使用的计算模型并

为计算模型提供一个代码生成器 (code generator) 或计算模型解释器 (interpreter for the MoC) (或者两者兼有)。“42” (Maraninchi and Bhouhadiba, 2007) 给出了另一种替代方法, 它在模型中集成了一个定制的计算模型。

补充阅读: 支持异构建模的工具

很多被广泛使用的工具都提供了一些计算模型的固定组合。商业工具包括, MathWorks 的 Simulink/StateFlow (将连续和离散时间角色模型与有限状态机相结合)、美国国家仪器公司的 LabVIEW (将数据流角色模型有限状态机和一个时间驱动的计算模型相结合)。Statemate (Harel et al., 1990) 和 SCADE (Berry, 2003) 将有限状态机和一个同步/响应体系 (synchronous/reactive formalism) 进行结合。Giotto (Henzinger et al., 2001) 和 TDL (Pree and Templ, 2006) 将有限状态机和时间驱动的计算模型进行结合。有些混合系统建模和仿真工具结合了连续时间动态系统和有限状态机 (Carloni et al., 2006)。

Y-chart 方法支持异构建模, 是一种比较流行的软硬件协同设计方法 (Kienhuis et al., 2001)。这种方法将硬件实现的建模与应用程序行为的建模分开, 并提供将这两种分离的模型组合起来的机制。这些机制允许开发者在硬件开销和软件设计复杂度之间进行折中。Metropolis 是实现此目标的一个简洁工具 (Goessler and Sangiovanni-Vincentelli, 2002)。它引入一个“质量管理者”(quantity manager) 来调节理想功能和实现功能所必需资源之间的一种均衡。

从组件具有并发性并通过端口通信的概念上来说, Modelica (Fritzson, 2003; Modelica Association, 2009) 也有类似于角色的语义。但这里的端口并不是输入或输出, 而是端口之间的连接声明了变量间的等式约束。这种方法有明显的好处, 尤其是对于基于微分代数方程 (Differential-Algebraic Equation, DAE) 进行描述的物理模型来说。但是, 这种方法在组合不同种类的计算模型上显得吃力。

DESTECS (DEsign Support and Tooling for Embedded Control Software, 嵌入式控制的设计支持和工具化软件) 是一个由大学和工业界联合支持的工具, 该联盟的工作重心是容错嵌入式系统 (Fitzgerald et al., 2010)。该工具集成了 20-sim (Broenink, 1997) 的连续时间模型和 VDM (Vienna Development Method, 维也纳开发方法) 的离散事件模型。DESTECS 将时间同步化, 并在两个工具中进行变量传递。

1.7 时间模型

有些计算模型有时间 (time) 概念。具体来说, 这意味着角色间的通信和角色所执行的计算是在一个逻辑时间轴上的。更具体地说, 这意味着将有这样一个概念: 两个动作 (通信和计算) 要么按时间顺序发生 (一个发生在另一个之前), 要么同时发生 (并发)。另外时间概念也可能有个度量工具, 这意味着两个动作之间的时间间隔可以被测量。

Ptolemy II 为互操作性提供的一种很关键的机制就是时间概念的一致性。甚至当我们将不涉及时间概念的模型 (比如数据流模型和有限状态机模型) 和对时间概念有很强依赖的模

型（比如离散事件模型和连续时间模型）进行结合时，该机制仍然有效。本节将概述这种机制的关键特征。

1.7.1 层次化时间

在 1.5 节和 1.6 节讨论的层次结构模型是时间管理的核心。通常，只有顶层指示器进行时间的推进。模型中的其他指示器从闭合指示器（enclosing director）中获取当前的模型时间。如果顶层指示器没有实现一个计时模型，那么时间不会被推进。因此，计时模型通常包含了一个已实现计时模型的指示器。

计时和不计时的计算模型可能在层次结构中是交叉的。然而，在稍后的讨论中确实有些不合理又特别有用的组合，特别是第 8 章中讨论的模态模型。

在一个模型中，时间的推进可能是不同的。在第 8 章的模态模型中，时间的推进可以在子模型中临时暂停（Lee and Tripakis, 2010）。更为普遍的是，按照第 10 章的解释，在层次结构的不同层中时间会按不同速率推进。该特征在分布式系统的建模中特别有用，说明在分布式系统中要保持完全一致的统一时间在物理上是不可能的。因为存在时间多样性（multiform time），所以模型只能高度逼近真实，即明确承认时间只能被不完美地测量。

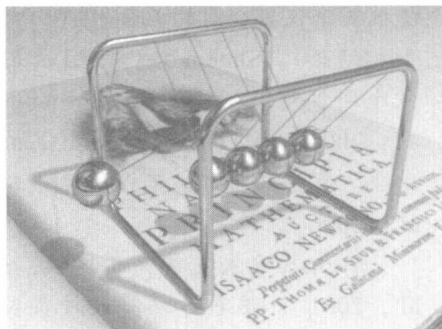


图 1-5 牛顿摆，图由 Dominique Toussaint 提供，在 GNU 自由文档许可条款下可用，版本为 1.2 或更高

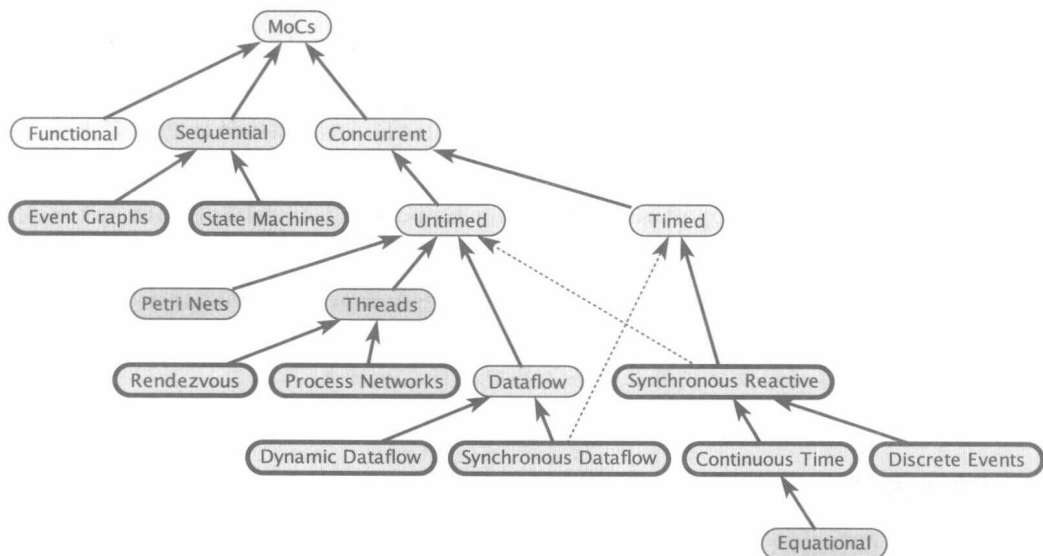


图 1-6 汇总计算模型之间的关系（本书详细介绍了加粗框内的模型）

1.7.2 超密时间

除了提供多样时间（multiform time）模型外，Ptolemy II 还提供了一种称为超密时间（superdensetime）的时间模型（Manna and Pnueli, 1993；Maler et al., 1992；Lee and Zheng,

2005; Cataldo et al, 2006)。超密时间的值是一值对 (t, n) , 称为时间戳 (timestamp), t 是模型时间, n 是微步 (microstep, 也称为索引 (index))。模型时间代表事件发生的时间, 微步代表在相同模型时间上发生事件的顺序。即使 $n_1 \neq n_2$, 两个时间戳 (t, n_1) 和 (t, n_2) 也能解读为同时 (simultaneous) (在很弱的意义下)。较强意义的同时性 (simultaneity) 需要时间戳完全相等 (模型时间和微步都相等)。通过下例来阐述超密时间如何取值。

例 1.3 为了理解微步的作用, 首先来考虑牛顿摆, 如图 1-5 所示, 有一个在细绳上悬挂 5 个钢球的玩具。如果抬起第一个球并释放它, 它会撞击第二个球, 但第二个球不会动而第五个球会上升。

考虑第二个球的动量 p , 将其视为时间的函数。第二个球不会移动, 因此它的动量必须处处为零。

但是第一个球的动量是通过第二个球转移到第五个球, 所以动量不可能总是零。令 \mathbb{R} 代表实数。令 $p: \mathbb{R} \rightarrow \mathbb{R}$ 代表第二个球的动量函数, 令 τ 表示撞击发生的时间。于是

$$p(t) = \begin{cases} p & t = \tau \\ 0 & \text{其他} \end{cases} \quad (1-1)$$

对一些常量 P 和所有的 $t \in \mathbb{R}$ 。在时间 τ 的瞬间前后, 球的动量都为零, 但是时间 τ 上, 球的动量和速度成正比, 所以

$$p(t) = Mv(t)$$

其中 M 是球的质量。因此, 结合式 (1-1),

$$v(t) = \begin{cases} P/M & t = \tau \\ 0 & \text{其他} \end{cases} \quad (1-2)$$

物体的位移是它速度的积分:

$$x(t) = x(0) + \int_0^t v(\tau) d\tau$$

其中 $x(0)$ 是初始位移。在任意时刻 t 式 (1-2) 给出的函数的积分都为零, 因此球不会移动, 尽管那个瞬间球的动量不为零。

上述物理模型的主要工作是描述物理现象, 但是有两个缺陷。首先, 它违反了动量守恒定律的基本物理原理。在碰撞的瞬间, 中间的三个球都会同时获得非零动量, 所以看起来总动量是神奇地增加了。第二, 该模型不能直接转换为一种离散表示。

对于一个信号来说, 它的离散表示是指其值按照时间顺序分散排列。(数学细节请参看 9.2.1 节的补充阅读)。任何如式 (1-1) 的动量表示和式 (1-2) 的速度表示都是模棱两可的。如果序列不包含碰撞时间上的值, 那么这种表示就无法捕捉到动量是通过球来转移这一事实。如果序列包含碰撞时间上的值, 那么这种表示就不能与在某段时间间隔上有非零动量的信号表示相区分, 由此会建立移动球的模型。在这样的离散表示下, 没有语义能区分瞬时事件和迅速变化的连续事件。

超密时间能解决这两个问题。具体来说, 第二个球的动量可以明确地表示为一系列的样本, 其中 $p(\tau, 0) = 0$, $p(\tau, 1) = P$, $p(\tau, 2) = 0$, τ 表示碰撞的时间, 第三个球只有在超密时间 $(\tau, 2)$ 才有非零动量。在碰撞时, 每个球最初动量为零, 然后为非零, 然后又变成零, 所有状态瞬间完成。对于中间的三个球来说, 有非零动量的事件是弱同时性的, 但不是强同时性的 (strongly simultaneous)。这样动量是守恒的, 模型是明确、无歧义和离散的。

有人会说物理系统实际上不是离散的。即使做工精良的钢球也会发生形变，因此碰撞实际上是连续的过程，而不是离散事件。虽然该话是正确的，但是在建模时，并不希望进行形式主义的建模来强行建立一个“详细的模型”而非“适当的模型”。这样一个“详细的”牛顿摆模型将会非常复杂，并产生非常难以分析的非线性动态力学分析。这样，模型的保真度(fidelity)会提高，但是它的可理解性和可分析性却相当低。

上面的例子表明，一个包含瞬时事件的物理过程最好采用 $p: \mathbb{R} \times \mathbb{N} \rightarrow \mathbb{R}$ 的形式来建模，而不是更为传统的 $p: \mathbb{R} \rightarrow \mathbb{R}$ 来建模，其中 \mathbb{N} 是自然数。后者用于对连续过程建模是足够的，但对离散事件建模就有所不足。在任意时间 $t \in \mathbb{R}$ ，信号都有一个值序列，该序列是根据微步来排序的。这个信号不能被理解为一个迅速变化的连续信号。

如果 $t_1 = t_2$ ，也就是说两个时间戳 (t_1, n_1) 和 (t_2, n_2) 是弱同时性的(weakly simultaneous)，此外，如果还有 $n_1 = n_2$ ，那就是强同时性的(strongly simultaneous)。

这样就可以对因果相关，但具有弱同时性事件进行表示在时间戳 (t, n_1) 和 (t, n_2) 上其中 $n_1 \neq n_2$ ，信号可能包含两种不同的事件。信号可能因此包含了具有弱同时性但又彼此不同的事件。两个不同信号可能包含强同时性的事件，但一个单独信号不可能包含两个不同的强同时性的事件。这个时间模型明确无歧义地表示了离散事件、连续时间中的非连续信号以及离散信号中的瞬时事件(zero-time event)序列。

超密时间是按字典序(lexicographically)排列的(如同字典)，这意味着如果 $t_1 < t_2$ 或者 $t_1 = t_2, n_1 < n_2$ ，那么 $(t_1, n_1) < (t_2, n_2)$ 。因此，如果一个事件比另一个事件的模型时间少，或者在相同模型时间下微步较低，那么该事件被认为发生在另一个事件之前。在 Lee and Sangiovanni-Vincentelli (1998) 的标签信号模型(tagged-signal model)中，时间戳是标签(tag)的一种特殊实现。

1.7.3 时间的数字表示

计算机不能完美地表示实数，所以 $(t, n) \in \mathbb{R} \times \mathbb{N}$ 形式的时间戳无法实现。许多软件系统利用双精度浮点数来近似表示时间 t 。但是这种表示方式有两个严重的弊端。第一，数字的精度(precision)(可表示的两个数字之间的最小距离)取决于其自身的数量级。因此，在这样的系统中，随着时间的增加，表示时间的精度会下降。第二，在这种表示方法中，加法和减法会引入量化误差，所以 $(t_1 + t_2) + t_3 = t_1 + (t_2 + t_3)$ 不一定是对的。这会极大削弱同时性(simultaneity)概念的语义，因为两个事件是否可以认为是“同时的”(无论是弱同时还是强同时)，这依赖于时间戳的计算方式。

Ptolemy II 通过将时间分辨率(time resolution)变为单一的全局常数解决了这个问题。给出模型时间为 $t = mr$ ，其中 m 是任意的大整数，时间分辨率 r 是双精度浮点数。倍数 m 是一个 Java 的大整数(任意的大整数)，因此它永远不会溢出。时间分辨率 r 是模型所有指示器的共享参数。因此，一个模型不论有多大的时间规模，它的整个分层结构和执行过程都会有相同的时间分辨率。此外，加法和减法的时间值不会受到量化误差的影响。在默认情况下，时间分辨率 $r = 10^{-10}$ ，它代表 1/10 纳秒。比如，倍数 $m = 10^{11}$ ，那么时间 t 就是 10 秒。

在 Ptolemy II 中，时间戳 (t, n) 中的微步 n 由一个 32 位的整数表示。因此微步很容易溢出。该溢出可以通过在第 7 章讨论的震颤芝诺行为的模型来消除。

1.8 域和指示器概述

在 Ptolemy II 中, 计算模型的实现称为域[⊖]。本节将简要描述 Ptolemy II 中实现的域, 但没有涵盖所有的域。本节主要目的是展示计算模型的多样性。这些域以及其他域将在接下来的章节中得到更详细的介绍。图 1-6 总结了这些域之间的关系。

这里所描述的所有域都具有确定性, 除非模型明确地说明具有非确定性行为。即如果需要不确定性, 则要在模型中明确地加入不确定性; 因此不确定性不会因为模型框架的弱语义而意外发生。如果符合以下条件, 则可称一个域是确定性的: 即角色之间发送的信号, 包括消息所传递的数据值、消息顺序和时间戳, 不受调度决策的影响 (尽管模型有并发性)。保证确定性在并发计算模型中相当困难, 并且提供合理的不确定机制也是相当具有挑战性的。目标是保证当一个模型包含不确定性的行为时, 它应被模型的建立者明确地描述; 不确定性行为不会意外出现, 并且出现也不会让用户感到意外。

数据流 (dataflow)。Ptolemy II 包含多个数据流域, 将在第 3 章中描述。数据流域中角色的执行包含一系列点火 (firing) 行为, 每一次点火行为都是对输入数据有效性的反应。一次点火行为就是一次 (很小的) 计算, 消耗输入数据, 产生输出数据。

同步数据流域 (Synchronous DataFlow, SDF) (Lee and Messerschmitt, 1987b) 是很简单的, 也几乎是使用最多的域。在同步数据流域中, 当一个角色执行时, 它消耗固定数量的输入数据, 并对每一个输出端口产生固定数量的输出数据。同步数据流域的一个好处 (如第 3 章中描述的那样) 就是可以静态地检查潜在的死锁和有界性, 并且可以静态地进行调度的计算 (包括并行调度)。这个域中的通信由固定容量的先进先出 (First In First Out, FIFO) 队列实现, 并且组件的执行顺序被静态地调度。尽管同步数据流可以是计时或不计时的, 但通常它是不计时的, 如图 1-6 所示。

相反, 动态数据流 (Dynamic DataFlow, DDF) 域比同步数据流域更为灵活, 它进行在线的调度策略计算。在动态数据流中, FIFO 队列的容量是无界限的。当角色之间的通信方式取决于两者之间传递的数据时, 动态数据流是十分有用的。

数据流模型对于表示流系统 (streaming system) 是十分理想的, 因为流系统中组件间数据值顺序的流动也是相对有规律的。比如说, 应用到信号处理系统中是特别适合的 (如音频和视频系统)。

进程网络。在进程网络 (Process Network, PN) 域中, 如第 4 章描述的那样, 角色表示那些通过 FIFO 队列 (概念上无限容量的) 进行通信的并发进程 (Lee and Parks, 1995)。写入队列通常是立即成功的, 但从空队列里读取内容会导致读取阻塞。简单的阻塞读取和非阻塞写入策略保证了模型的确定性 (Kahn and MacQueen, 1977)。然而, 本文将模型进行了扩展以支持特定形式的不确定性。每个角色都在自己的 Java 线程上执行, 所以对于多核机器, 它们可以并行执行。进程网络域实现了数据流的一般化, 该域是不计时的。角色表现为连续的执行过程以替代离散的点火 (Lee and Matsikoudis, 2009)。

进程网络域很适合表示通过消息传递进行通信的并发过程。最终, 消息将以发送的顺序被交付。消息的传递被假定为可靠的, 所以发送者不用期待或接收任何确认消息。这种域为“无反馈通信” (send and forget)。

⊖ 术语“域”来自于天体物理学的一个想象概念, 即宇宙中遵循不同物理规律的区域。一个计算模型展现了它所管理的子模型的“物理规律”。

进程网络也提供相对简单的方法来实现模型的并行执行。每个角色在自己的线程中执行，大多数现代的操作系统都能自动将线程映射到可用的核上。注意，如果角色的构造相对精细，也就是说它们对每次通信执行很少的计算，则多线程和内部线程通信的开销可能会掩盖并行执行的性能优势。这样，建模者只能在粗粒度模型中考虑并行性能优势。

会话。会话域 (rendezvous domain) 在第 4 章详述，它和进程网络相似，因为角色都表示并发进程。但是，与进程网络的“无反馈通信”不同，在会话域中，角色通过原子瞬时值数据交换进行通信。当一个角色向另一个角色发送消息时，在接收者准备接收之前，发送者会阻塞发送。相似地，若一个角色试图读取输入数据，当发送者准备发送之前，接收者会阻塞接收。这样的结果就是，首先到达会话点的过程会暂停以等待另一个过程也到达会话点 (Hoare, 1978)。在这个域中可以创建多路会话进程，在多个进程都到达会话点之前，任何进程都不能继续执行。与进程网络一样，这个域也是不计时的，它支持明确的不确定性，并且可以透明地利用多核机器。

会话域在解决异步资源竞争问题时十分有用，这种情况是指单一资源被多个异步过程共享的情况。

同步响应。同步响应 (Synchronous-Reactive, SR) 域在第 5 章详述，它基于同步语言的语义 (Benveniste 和 Berry, 1991; Halb-wachs et al., 1991; Edwards and Lee, 2003a)。同步语言的准则很简单，尽管结果意义重大。它的执行遵循全局时钟的节拍 (tick)。每个节拍上，每个变量 (在 Ptolemy II 中图示为连接块与块之间的线) 可能会得到一个值，也可能不会。它的值 (或缺失) 是由一个其输出端口连接在线上的角色赋给的。这个角色实现了这样的功能：将它输入端口的值映射到输出端口 (在每个节拍上，功能有所变化)。比如，在图 1-7 中，在一个特定节拍上的变量 x 和 y 有如下关联：

$$x = f(y), \text{ 且 } y = g(x)$$

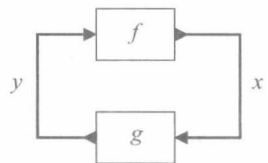


图 1-7 一个简单的反馈系统

域中指示器的任务是在每个时钟节拍上寻找满足上式的 x 和 y 的值。这种解法称为**定点求解** (fixed point)。尽管同步响应模型是默认为不计时的，但是当节拍之间有固定时间间隔[⊖]时，它可以是计时的。

同步响应域和数据流域、进程网络域是相似的，因为角色彼此之间发送数据流。但不同的是，同步响应域中的流是同步的；在时钟的一个节拍上，每一个通信链路要么有一个消息，要么消息明确是缺失的。相比之下，数据流模型更具异步性；一个消息是“缺失的”，仅仅是因为发生调度意外而导致其未到达。为了阻止不确定性，进程网络和数据流都没有“缺失的”输入这一语义概念。输入总是有消息的 (或者将有消息的，这种情况下角色被要求等待消息到来)。

同步响应很适合更复杂的控制流，即一个角色可以依据一个消息是否出现而采取不同动作。通过同步动作，域可以掌控那些不含非确定性的场景。同步响应相比数据流和进程网络较少出现，因为每个时钟节拍必须紧密连贯。所以，它较难并行执行。

有限状态机。有限状态机 (Finite-State Machine, FSM) 域在第 6 章详述，它是本书讨论的各种域中唯一一个非并发性的域。这个域中的组件不是角色，而是状态，并且组件之间的联系不表示通信链路，而是状态之间的转移。转移通过检测器 (guard) 来确定状态转移在

⊖ 如果需要一个可变的节拍之间的时间间隔，可以将 SR 模型加入 DE 模型中来实现。

何时发生。

有限状态机可以用来定义其他域中使用的角色的行为。角色有很多输入和输出。当角色执行时，有限状态机读入输入，评估检测器得到结果以决定执行哪种转移（transition），然后按照选择的转移指定的值进行输出。有限状态机也可以有局部变量，它们的值可以被转移修改（提供了一种称为扩展状态机的计算模型）。

有限状态机也可以用来创建一大类称为模态模型（modal model）的层次模型，这在第8章中讨论。在模态模型中，有限状态机的模型包含子模型，这些子模型处理输入且产生输出。有限状态机的每一个状态表示一种执行模式，且模式细化（mode refinement）定义了该模式下的行为。模式细化是一个子模型，有自己的指示器，且仅当有限状态机在相应状态时它才是活动的。当一个子模型不在活动状态时，它的本地时间不再推进，如1.7.1节解释的那样。

离散事件。在离散事件（Discrete Event, DE）域中，如第7章所述，角色通过位于同一时间轴的事件进行通信。每个事件都有一个值和时间戳，并且角色对事件的处理是按照时间顺序的。由角色产生的输出事件在时间上不得早于被消耗的输入事件。也就是说，离散事件域中角色之间是因果联系的。

该模型按照一个全局事件队列进行执行。当一个角色产生一个输出事件时，该事件根据其时间戳被插入队列。在离散事件域模型的每次迭代中，最小时间戳的事件从全局事件队列中被删除，且它们的目标角色被点火。离散事件域支持同时性（simultaneous）事件。当一个角色被点火时，指示器就计算一个定点，该过程与同步响应类似（Lee and Zheng, 2007）。离散事件域和众所周知的离散事件系统规范（Discrete Event System Specification, EDVS）的形式体系紧密相关（Zeigler et al., 2000），EDVS广泛应用于大型复杂系统的仿真。Ptolemy II 多样的 DE（Discrete Event）语义由 Lee（1999）给出。

DE 很适合对时间相关的复杂系统进行建模，例如网络系统、数字硬件电路、金融系统、行政管理系统等。另外，第10章还将介绍 DE 如何被扩展应用到多样时间（mutiform time）系统中。

连续时间。连续时间（continuous time）域（Lee and Zheng, 2005）（在第9章详述）对常微分方程（Ordinary Differential Equation, ODE）进行建模，同时它也支持离散事件。表示积分器（integrator）的特殊角色被连接在反馈回路中，用以进行常微分方程的表示。域的每个连接（connection）代表一个连续时间函数，组件表示函数之间的关系。

连续模型使用数值方法计算常微分方程的解。与同步响应和离散事件一样，在每一个瞬间，指示器为所有信号值计算一个固定点（Lee and Zheng, 2007）。在每次迭代中，时间以一定增量向前推进，这个增量取决于 ODE 求解器。为了推进时间，指示器在求解器的帮助下进行时间戳的选择，然后通过该时间步长进行角色的推测执行。如果时间步长足够小（比如一些关键事件：层间交叉，模式改变，等行足够点火次数和迭代次数计算等），那么指示器提交时间增量。

连续指示器与 Ptolemy II 中所有的计时域都可交互。它与有限状态机结合产生了一种特定的模态模型，叫作混合系统（hybrid system）（Lee and Zheng, 2005; Lee, 2009）。将它与同步响应域或离散事件域结合同样比较有用。

Ptera。Ptera 域在第11章详述，实现了事件图（event graph）的一个变体。在 Ptera 中，组件不是角色。这里的组件是事件，组件之间的关系决定事件之间的关系。一个 Ptera 模型

展示系统中的一个事件是怎么触发另一些事件的。Ptera 是一个计时模型，与有限状态机一样，它可用于定义另一个域的角色行为。另外，事件是可以组合的，因为它们有一些与它们相关的动作，这些动作本身就是被一个子模型定义的，而这个子模型使用了另一个域描述。Ptera 在描述计时行为时很有用，这些行为中，输入时间有可能触发一系列反应。

1.9 案例研究

CPS 本质上是异构的。因此，CPS 模型的特点是能够与计算模型结合。本节将通过一个实例来展示多个计算模型的使用。这是一个高度简化的实例，但只需要一点点想象力，就能很容易地看出一个模型如何发展成一个准确和完整的大型复杂系统模型。关于大型复杂系统，我们很容易想到的一个例子是加载在智能电网或运载工具（如飞机和高级汽车）上的电力系统。这种系统具有多种电力来源（如风力发电机、太阳能电池板、涡轮机、备用发电机等），应当协调地为多样性的负载提供能源。该系统包括控制器和监督控制器，其中控制器用来使发电机的电压与频率保持在近似的常数，监督控制器则用来提供负载和发电机的连接与断开服务，并提供故障处理服务以保护设备。该系统还包括网络，该网络的动态变化能够对整个系统产生影响。

下例所示为一个高度简化版本的系统，用来展示如何使各种车辆运行管理系统发挥作用。

例 1.4 一个燃气发电机的简化模型，如图 1-8 所示，它能够连接和断开负载。这是一个连续时间模型，由相应的连续时间域指示器（Continuous Director）表示，详见第 9 章。该模型有两个输入，一个驱动（drive）信号和一个负载导纳（loadAdmittance）。输出是一个电压（voltage）信号。此外，该模型有 3 个参数，一个时间常量 T ，一个输出阻抗 Z 和一个驱动极限 L 。当发电机获得或多或少的燃气（由 drive 输入表示）且负载变化（由 loadAdmittance 输入表示）时，模型将随时间给出一个变化的发电机电压输出。

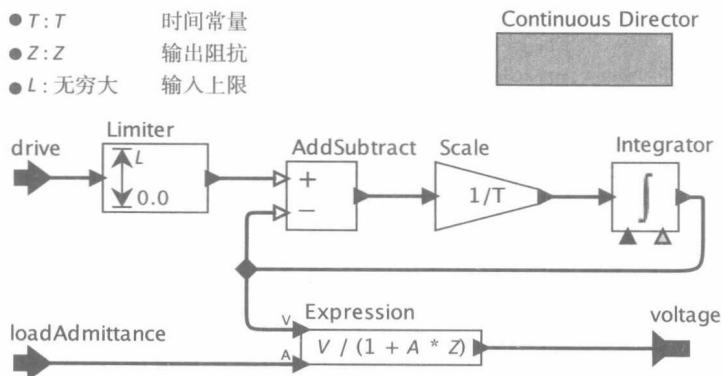


图 1-8 燃气动力发电机简化模型

该模型展示了简化的线性和非线性的动力系统。非线性动力由限制驱动（drive）输入的 Limiter 角色实现（见 2.2 节补充阅读：Math 库）。特别是，如果 drive 输入变成负数，它将设置驱动为零（不可能将燃气从发电机里抽出来）。当在参数 L 所给的上界时，它使驱动（drive）输入达到饱和。其中 L 的默认值为无穷（Infinity），这意味着不设饱和值（发电机能够承受任意大的驱动（drive）输入）。

本模型中，线性动力部分是由一个小的反馈回路提供的，其中包括一个 AddSubtract 角色、一个 Scale 角色，以及一个 Integrator 角色。如果 Limiter 角色的输出是 D ，那么该回路提供一个值 V ，并满足如下微分方程：

$$\frac{dV}{dt} = \frac{1}{T}(D - V)$$

其中 D 和 V 都是时间的函数（关于模型如何产生上式方程详见第9章）。

对于该方程的理解不是此处重点，因为模型的这个部分通常由机械工程师来构建，他们是该方面的专家。但是读者可以做一些直观的观察。第一，如果 $D=V$ ，那么导数为零，所以发电机是稳定的并产生一个恒定的输出。第二，当 $D \neq V$ 时，反馈回路将调整 V 的值，使其接近 D 。如果 $D > V$ ，那么方程使 V 的导数为正，那么 V 会增加。如果 $D < V$ ，那么导数为负，所以 V 会减少。事实上，输出 V 会在时间常量 T 内指数收敛到 D 。**时间常量**（time constant）是指一个指数信号达到最终（渐进）值的 $1-1/e \approx 63.2\%$ 所需的时间。

模型的最后一部分是对负载产生的影响进行建模。这种影响可由 Expression 角色来进行建模（见 13.2.4 节），利用欧姆定律计算输出电压的函数值 V ，该函数体现发电机的效率、输出阻抗 Z 和负载导纳 A 。电气工程师会认为该计算实现了一个简单的分压器。

在文本中，注意以下几点就足够了：如果 $A=0$ （无负载）或者 $Z=0$ （该发电机是理想电源），那么输出电压等于 V 。然而，真实的发电机并没有非零阻抗。当负载导纳从零开始增加时，输出电压会下降。

上面的模型是一个最简单有趣的连续动态模型。为了把该模型与数字控制器集成在一起，可以将模型封装成另一个定义了离散接口的模型，如下所述。

例 1.5 将图 1-8 所示的发电机模型进行封装，以提供一个离散接口，如图 1-9 所示。这里，驱动（drive）和负载导纳（loadAdmittance）输入进入 ZeroOrderHold 角色的实例。这样，输入信号就可以以离散事件的形式提供，而不是以连续时间信号的形式。ZeroOrderHold 角色在事件到达的间隙里将保持值不变，通过这样的方式将离散事件转换为连续时间信号（见 9.2 节）。

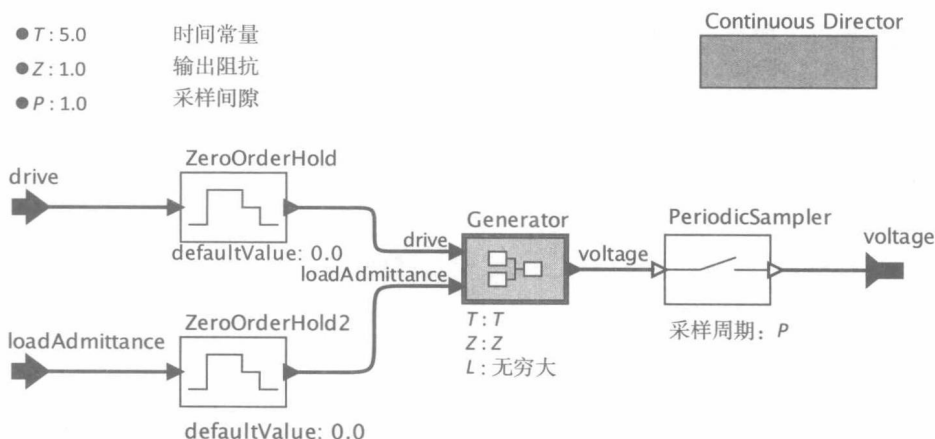


图 1-9 图 1-8 中发电机模型的封装，提供一个离散接口

输出电压经过可以产生离散事件的周期采样（PeriodicSampler）角色（见 9.2 节）。这里 Periodic Sampler 角色产生的离散事件是输出电压的采样值，采样周期为参数 P 。

该模型反提供了时间常数 T 和输出阻抗 Z ，但是隐藏了驱动极限 L 。当然，模型设计者可以选择要提供的参数。

一个连续时间模型可以嵌入于离散事件模型中（见第 7 章），如下所述。

例 1.6 图 1-9 中的离散发电机模型嵌入于图 1-10 中的离散事件模型中。该模型有两个参数，负载导纳 A 和过压阈值 OVT。DiscreteGenerator 角色的时间常数 T 设置为 5.0。该模型还包括其他两个组件，这两个组件：一个监视器（Supervisor），用来提供过载电压保护；另一个是控制器（Controller），它基于输出电压的测量值来调节 DiscreteGenerator 的 drive 输入。

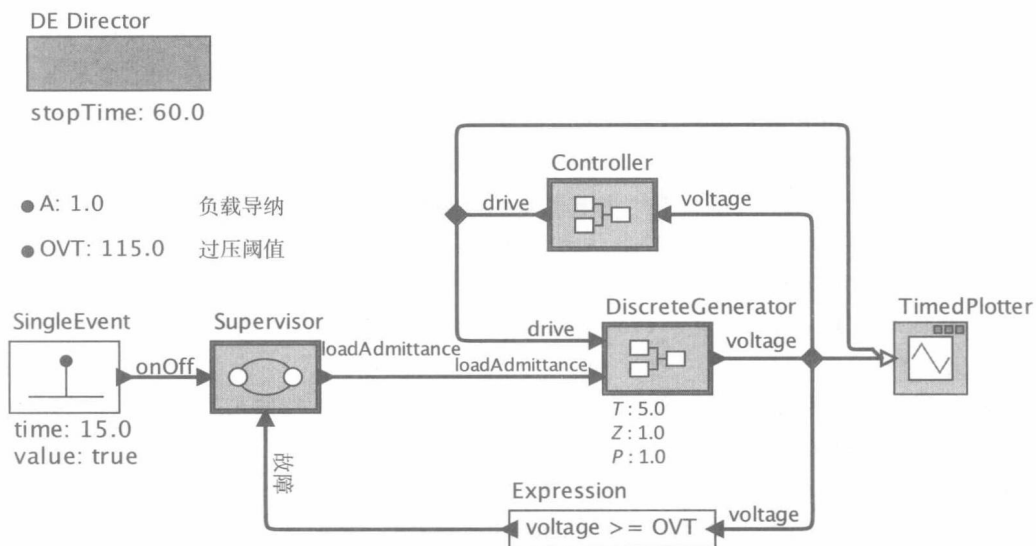


图 1-10 包含发电机、控制器、过压控制器的离散事件模型

此外，该模型还包括一个简单的测试场景，在该场景中，SingleEvent 角色（见 7.1 节补充阅读）在时间为 15.0 时请求一个负载连接，通过 TimedPlotter 角色（见第 17 章）进行模型运行结果的展示，如图 1-11 所示。

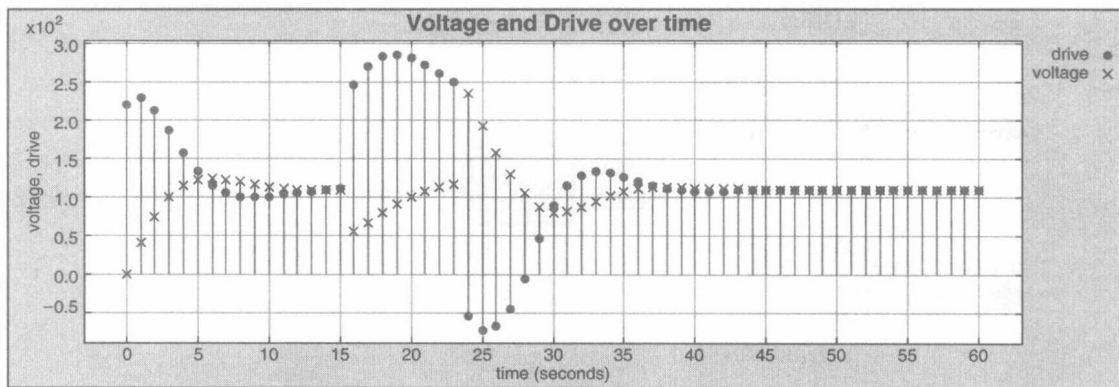


图 1-11 由图 1-10 中模型所生成的图

在测试场景中，与输出阻抗（1.0）相比，虽然负载导纳也相当高（1.0），但其会产生很

大的影响，因此，在时间为 15 秒接入负载时，电压突然下降为其目标值（110V）的一半。控制器（Controller）通过大幅增加驱动（drive）来为它补偿，但是这将导致电压超过目标值，在时间为 24 秒时，电压会超过 OVT 阈值。监视器（Supervisor）会通过断开负载对这种过压情况做出反应，因为现在发电机有了较大的 drive 输入。同时这导致电压上升得很高，控制器（Controller）通过调节驱动（drive）输入最终会使电压恢复到目标水平。

更进一步地注意到，当负载断开连接时，控制器使驱动（drive）信号为负。如果这是一个燃气发电机，控制器（Controller）将试图让发电机进行后向的燃气流动。幸运的是，发电机模型包含了个 Limiter 角色，它用来阻止模型真的提供负向的气体流。

图 1-10 中的模型包括了两种完全不同的控制器，一个监督用的控制器称为监视器（Supervisor）一个底层控制用的称为控制器（Controller）。这两个控制器被指定使用两个附加的计算模型（MoC），如下所述。

例 1.7：如图 1-12 所示，图 1-10 中的监视器（Supervisor）是一个有限状态机模型。虽然这里的符号将在第 6 章解释，但我们依然可以理解其一般行为。

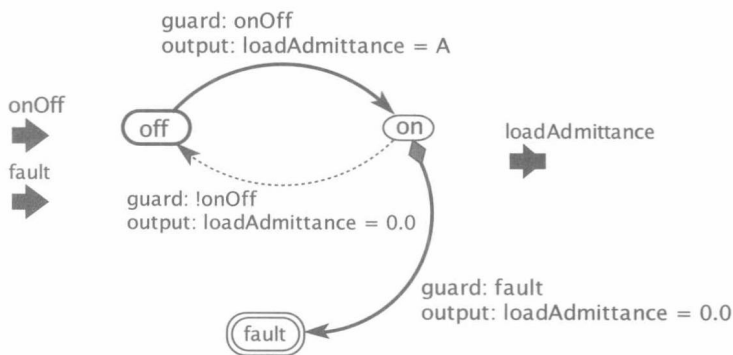


图 1-12 图 1-10 中的监视器（Supervisor）

这个有限状态机有两个输入：一个开关信号输入“onOff”（是一个请求连接或断开负载的布尔值），一个故障信号输入“fault”（是一个表示过压情况是否发生的布尔值）。有一个输出 loadAdmittance，它是提供给发电机的实际负载导纳。

有限状态机的初始状态是 off。当 onOff 输入为真，则有限状态机将从 off 状态转移到 on 状态，并输出一个 loadAdmittance，其值由参数 A 给出。它连接着负载。

当有限状态机的状态是 on 时，如果 fault 事件输入为真，则它将转移到最终状态 fault 并设置负载导纳（loadAdmittance）为 0.0，断开负载。如果 onOff 事件为假，那么它将转移到 off 状态，并且也会断开负载。这两种转移的不同在于一旦有限状态机进入 fault 状态，若没有系统重置（将有限状态机回复到初始状态）它将无法重连负载。

例 1.8 图 1-11 中的控制器（Controller）模型是图 1-13 所示的数据流模型。该模型使用 SDF 指示器（见第 3 章），它很适合进行采样数据信号处理。在这种情况下，控制器比较输入电压（voltage）和目标电压（110V），并将产生的误差信号输入 PID 控制器。PID 控制器是一种常用的线性时不变系统。控制工程师知道如何设定这种控制器的参数，但在该情况下，我们只需简单地选择一些实验性的参数来产生一个有趣的测试用例。

注意图 1-10 中模型有部分明显异构的，它涉及多个工程学科和计算机科学。一般来说，这种类型的模型都是工程师团队共同努力得出的结果，并且如果有一个框架可以使这些

团队将他们的模型组合起来，那么将是极具价值的。

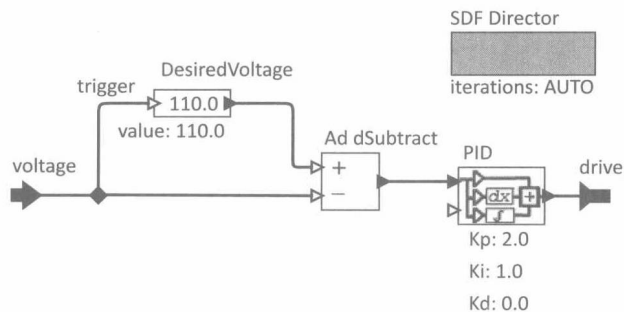


图 1-13 图 1-10 的控制器 (Controller)

关于该模型更多细节和扩展很容易想到，例如：

- 发电机可以定义成面向角色的类 (actor-oriented class)，以便它可以被重复应用，可以在一个集中的定义下开发和维护 (详见 2.6 节)。
- 使用第 9 章所讨论的技术，可以更精细地建立发电机模型，以反映更复杂的线性和非线性动态系统。
- 可以更精细地建立发电机模型，以考虑频率和相位的影响，例如通过使用相量的复数来表示阻抗和导纳。
- 大小可变的模型 (比如， n 个发电机和 m 个负载， n 和 m 是参数) 可以通过 2.7 节中所述的高阶组件来创建。
- 网络定时 (network timing)、时钟同步 (clock synchronization) 和共享资源竞争的影响可以用第 10 章中的技术来建模。
- 信号处理技术，如机器学习和频谱分析 (见第 3 章)，可以被集成到控制算法中。
- 单位系统 (units system) 模型可以包含在模型中，以使得模型对于时间、频率等测量单位的描述更加精确。
- 本体 (ontology) 可以加入模型中，使模型精确表示哪个信号和参数表示电压、导纳和阻抗等，或者使模型可以区分特定领域的概念，如发电机的内部电压和受输出阻抗和负载影响的输出端电压的区别。

1.10 小结

Ptolemy II 主要针对复杂系统领域中面向角色的建模，为并发和异构提供了一种规范的方法。层次化模型分解的核心概念是域，域可以实现特定的计算模型。技术上，一个域用于将组件之间的控制流和数据流与单个组件的实际功能分离。除了有利于层次化模型外，这种分离还能够显著地增加组件和模型的重用率。本书的余下部分将展示怎样建立 Ptolemy II 模型以及如何利用每个计算模型的性质。

图形化建模

Christopher Brooks、Edward A. Lee、Stephen Neuendorffer 和 John Reekie

本章主要介绍如何使用 Ptolemy 图形用户界面 (Graphical User Interface, GUI) Vergil 建立 Ptolemy II 仿真模型的教程。图 2-1 展示了在 Vergil 中建立的一个简单 Ptolemy II 模型。该模型显示在图形编辑器中, 这是 Ptolemy II 中多种可选的输入机制之一。例如, 也可在 Java 或 XML 中定义模型。

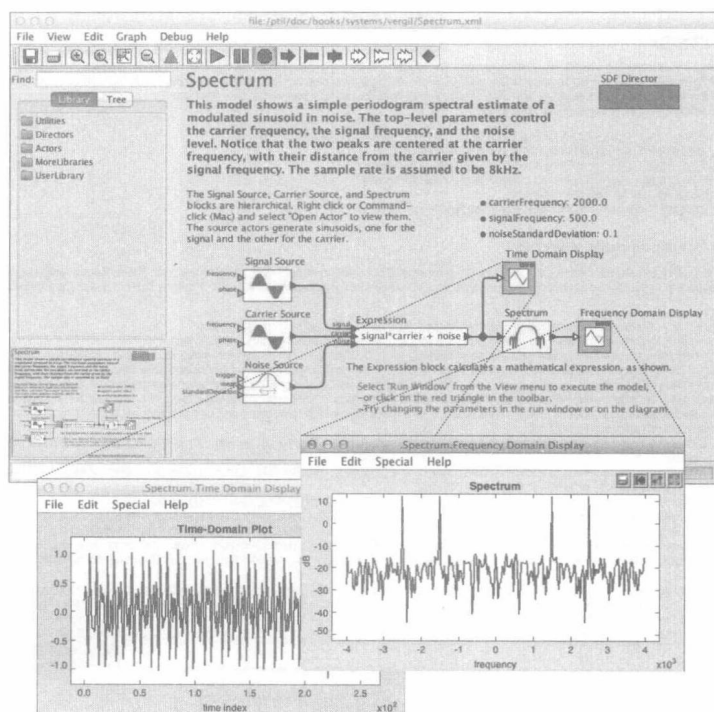


图 2-1 Vergil 窗口的例子以及运行模型的结果窗口

2.1 开始

执行本章示例需安装 Ptolemy II^①。一旦安装, 你将要调用 Vergil, 它将显示图 2-2 所示的初始欢迎窗口。“Tour of Ptolemy II”链接将带你进入图 2-3 所示页面。

① 见 <http://ptolemy.org/ptolemyII/ptIIlatest> 中最新版本, 在 <http://chess.eecs.berkeley.edu/ptexternal/> 可以访问最新开发中的版本。本书中的大多数图都附加有模型的在线版本, 使用任意浏览器均可。若你的机器支持 Java, 你可以通过单击一个链接 (这个链接使用了 Java Web Start) 来打开 Vergil 窗口, 不需要安装。你可以浏览、编辑、执行模型, 也可以将它保存到本地。Web Start 的信息请见 http://en.wikipedia.org/wiki/Java_Web_Start。

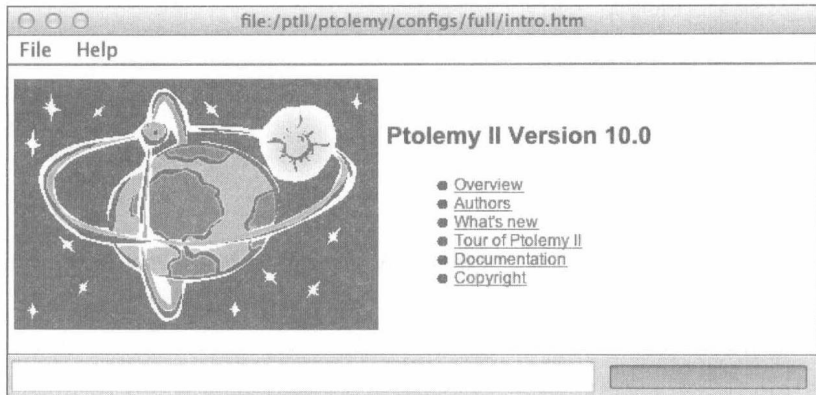


图 2-2 初始欢迎窗口

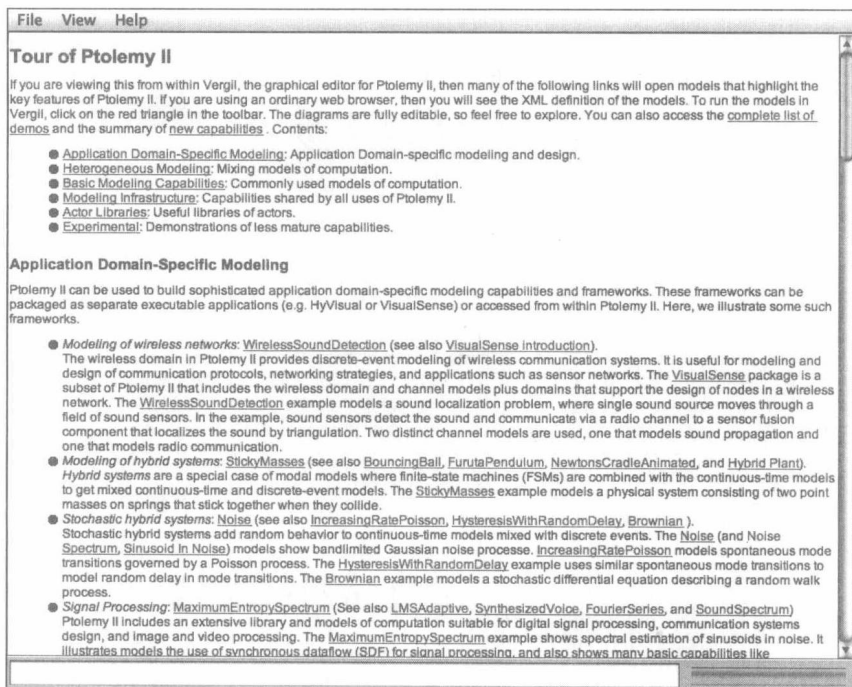


图 2-3 Tour of Ptolemy II 的页面

2.1.1 信号处理模型执行范例

进入“Tour of Ptolemy II”页面，在“Basic Modeling Capabilities”下列出的第一个例子（频谱分析）是图 2-1 所示的模型。该模型创建了一个正弦信号，加载正弦载波，添加噪声，然后估计功率谱。可通过工具栏中的 run 按钮（指向右边的蓝色三角形）执行该模型。如图 2-1 所示，两个信号图将在窗口显示。右图显示功率谱，左图显示时域信号。注意 4 个峰值，这表明是正弦调制。可通过双击图 2-1 窗口右上角附近的框图中的参数来调节信号和载波的频率，以及噪声的数量。

表达式（Expression）角色的图标（图形块）显示了表达式的计算：

```
signal*carrier + noise
```


表达式中的标识符——`signal`、`carrier` 以及 `noise` 指的是输入端口。表达式 (Expression) 角色是灵活的，它可以有任意数量的输入端口 (可以有任意的名称)，它使用了丰富的表达式语言来表示输入与输出之间的函数。它还可以进行模型参数的描述，这些表述式含在模型中。(表达式语言将在第 13 章详述。)

右击并选择 [Documentation → Get Documentation] 显示该角色说明文档。图 2-4 显示了 Expression 角色说明文档。

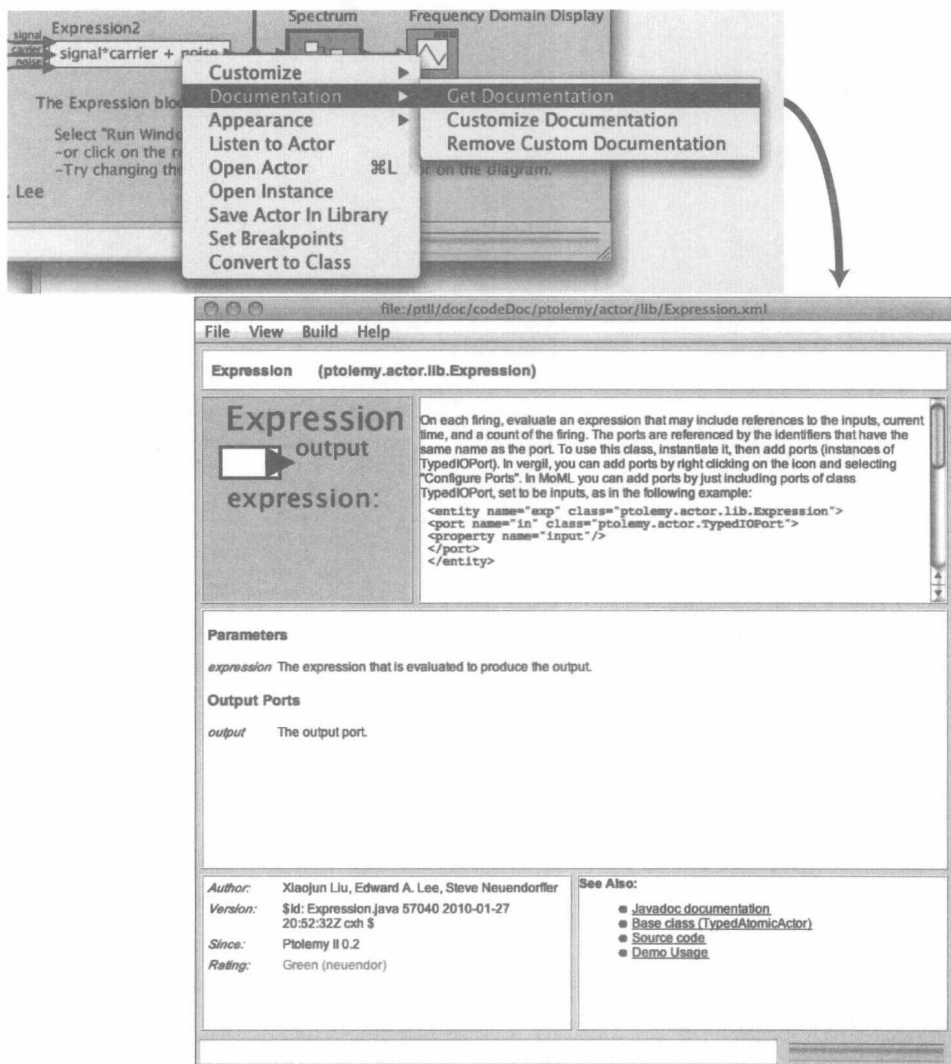


图 2-4 显示角色说明文档

图 2-1 中的 3 个角色都是复合角色 (composite actor)，这意味着它们的实现本身就是一个 Ptolemy II 模型。你可以调用 Open Actor 快捷菜单^①来查看 Signal Source 角色的具体实现，如图 2-5 所示。该图说明如何生成正弦信号。

① 快捷菜单是每个对象的专有菜单。你可以通过右击打开它 (没有鼠标右键的情况下，当鼠标在图标上方时单击控制面板)。

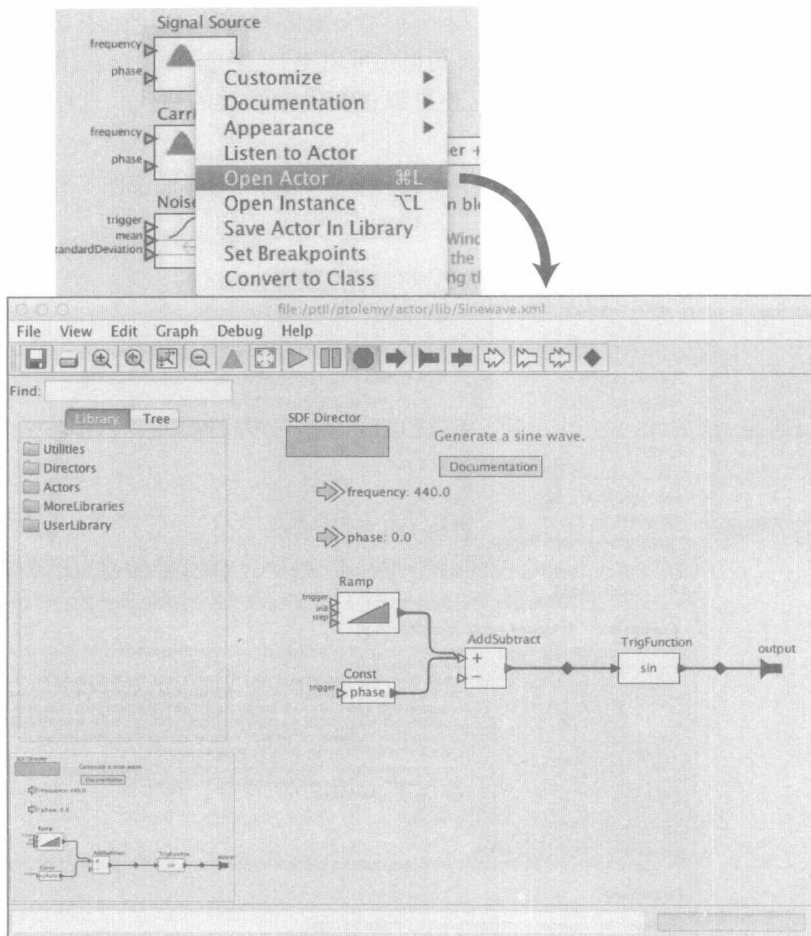


图 2-5 在复合角色上调用 Open Actor 来查看其具体实现

2.1.2 模型的创建和运行

通过选择菜单栏中的 [File→New→Graph Editor] 来创建一个新模型。如图 2-6 所示窗口。页面左侧显示了组件库，组件可以拖到模型建立区中。通过执行以下步骤创建一个简单的模型。

- 打开 Actors 库，然后打开 Sources。在 Sources→Generic 下找到 Const 角色并且拖动一个实例到建模区中。
- 打开 Sinks→GenericSinks 并且拖动一个 display 角色到页面中（见 2.1.3 节）。
- 从 Const 角色右边的输出端口拖曳出一个连接，连到 Display 角色的输入端口。
- 打开 Directors（指示器）库并且拖曳 SDF Director 到页面。指示器进行模型的功能控制，比如模型将要执行多少次迭代（默认情况下，只有一次迭代）。

完成以上步骤后将得到如图 2-7 所示的模型。在该模型中，Const 角色将创建一个字符串，Display 角色将显示这个字符串。通过双击或右击 Const 角色图标设置字符串变量为“Hello World”，并且选择 [Customize→Configure]，将显示图 2-8 所示的对话框。输入字符串的 value（参数）值“Hello World”（用引号标记）并单击 Commit（提交）按钮。（引号标记

保证表达式将表示为一个字符串。)如果运行该模型,窗口将显示文本“Hello World”。

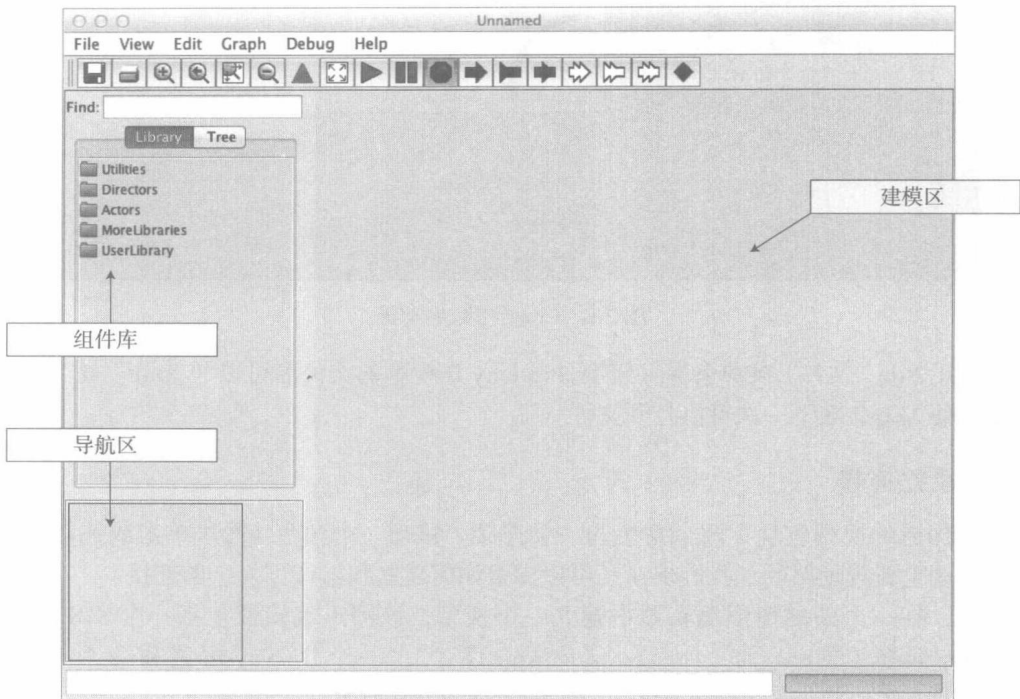


图 2-6 一个空的 Vergil 图形编辑器

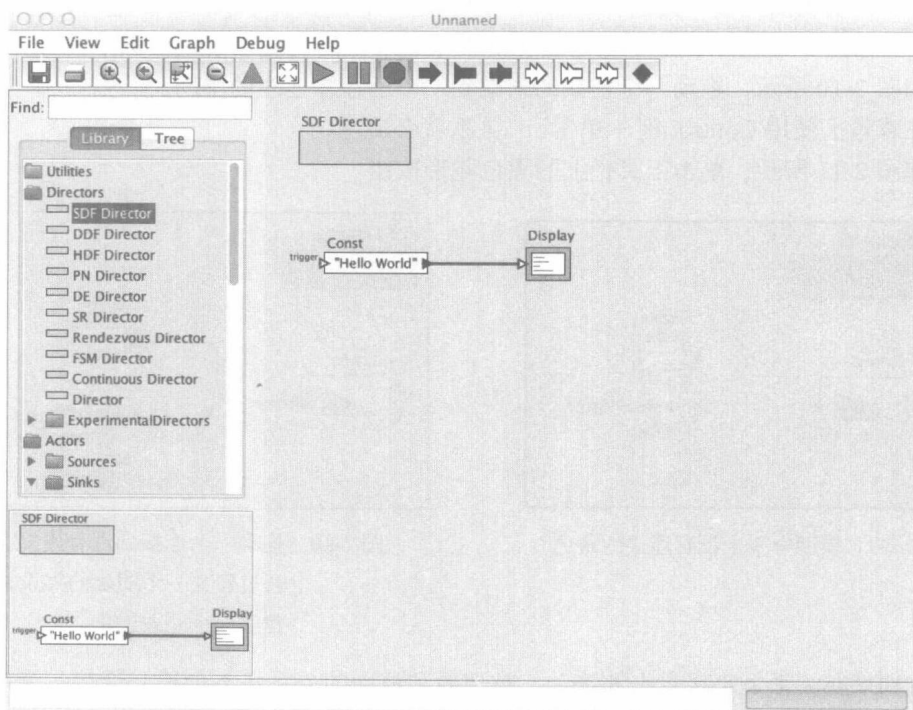


图 2-7 Hello World 示例

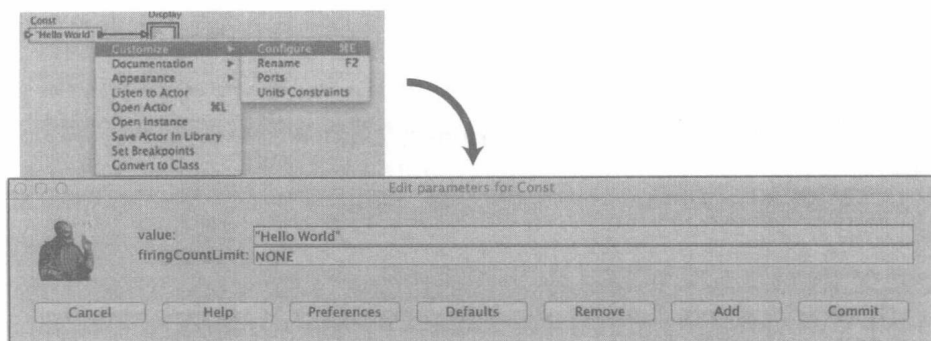


图 2-8 Const 参数编辑器

如果用 File（文件）菜单来保存模型，Ptolemy II 模型的文件名可以“.xml”或“.moml”作后缀以便 Vergil 在下次可以打开文件。

2.1.3 建立连接

上面构造的模型包含了两个角色和一个连接。移动一个角色（通过单击或拖动），连接就会自动重新规划路径。我们现在还可以探索如何建立和操作更复杂的连接。

首先，在一个新的图形编辑器中建立一个模型，该图形编辑器包括一个 SDF 指示器、一个 Ramp 角色（在 Sources → SequenceSources 库中）、一个 Display 角色和一个 Sequence-Plotter 角色（在 Sinks → SequenceSinks 库中），如图 2-9 所示。

假设 Ramp 角色的输出连接到 Display 角色和 SequencePlotter 角色。在这种情况下，图中需要有一个明确的关系。关系（relation）指的是一个分流器（splitter），它可以连接两个以上的端口。如图 2-11 所示，图中它由黑色菱形来表示。黑色菱形可以由以下几种方式创建：

- 如图 2-10 所示，拖动一个连接的末端到一个已存在连线的中间。
- 在背景上使用 Control 键 + 单击[⊖]，关系就会出现。
- 如图 2-11 所示，单击工具栏上的黑色菱形按钮。

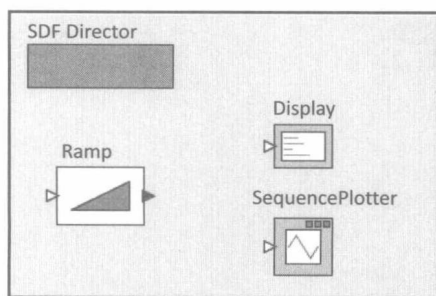


图 2-9 模型中 3 个没有连接的角色

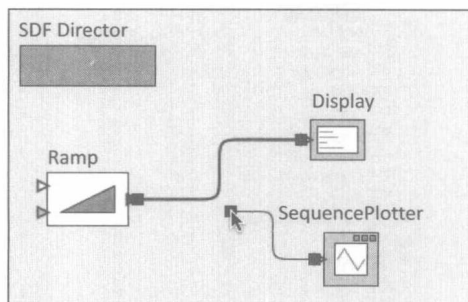


图 2-10 拖动一个连接到现有连接，
通过建立一个明确的关系来
建立一个三方连接

注意如果在关系上直接单击和拖动，被选择和移动的关系将不会建立连接。当单击和拖动关系时，需要按住 control 键才能建立连接。

⊖ 在苹果计算机系统中，Command+ 单击。本书中都沿用这种替换方法。

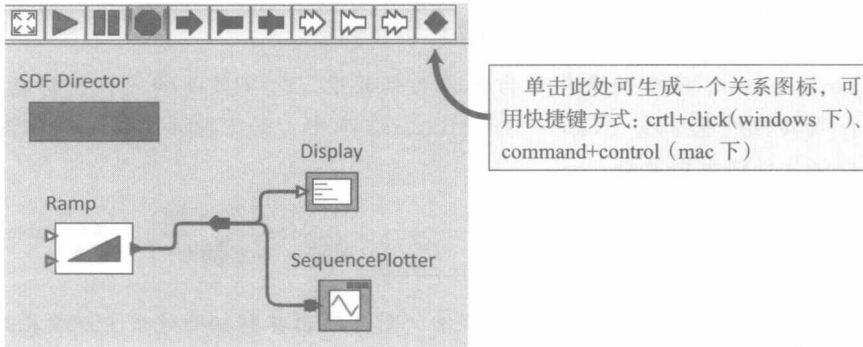


图 2-11 通过单击工具栏中的黑色菱形亦可建立关系

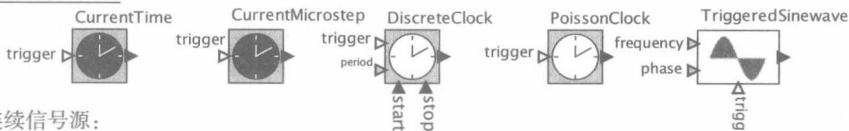
补充阅读：信号源

3 个 Actors → Sources 库中包含的信号源如下所示。

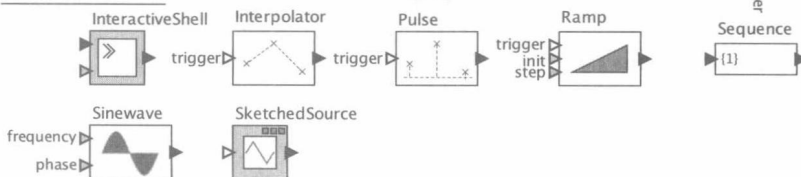
一般信号源：



计时信号源：



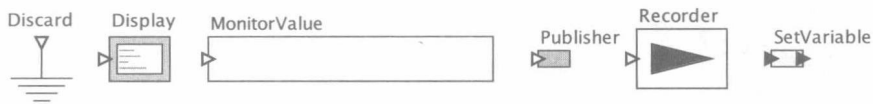
连续信号源：



- Const 通过参数设置（见第 13 章）来产生一个值（或表达式）。
- StringConst 产生字符串和引用其他参数（见 13.2.3 节）。
- Subscriber 输出从 Publisher 接收的数据。
- SubscriptionAggregator 使用指定的操作（目前为加法或乘法）把来自多个 Publisher 的数据整合。
- CurrentTime、CurrentMicrostep、DiscreteClock 和 PoissonClock 是第 7 章所描述的计时源。
- TriggeredSinewave 和 Sinewave 根据当前时间或特定的采样率分别产生正弦输出。
- InteractiveShell 将执行阶段打开的 Shell 窗口的输入值输出。
- Interpolator 和 Pulse 两者都产生波形，一个通过插值来产生，另一个通过在不同值间填充零得到。
- Ramp 产生一个稳步增加或减少的输出序列。
- Sequence 产生任意值序列。
- SketchedSource 产生一个可用鼠标指定序列。

补充阅读：Sinks 库

在 Actors → Sinks 库中的角色作为信号的目的地。大多数是绘图仪，将在第 17 章中描述。少数其他的包含在 Sinks → GenericSinks 库中，如下所示。这些接收器在它们的输入端口接收任何数据类型。



- **Discard** 丢弃所有输入。在大多数域中，断开输出具有相同效果（会话 Rendezvous 域是个例外）。
- 当模型执行时，**Display** 在新的文本窗口中进行输入值的显示。
- **MonitorValue** 在显示模型的 Vergil 图标中显示它的输入。
- **Publisher** 给 Subscriber 和 SubscriptionAggregator 的实例建立已被命名的连接（见 2.1.3 节）。
- **Recorder** 内部存储输入值，定制需要访问这些值的 Java 代码。
- **SetVariable** 设置定义在模型中的变量和参数的值。因为这个变量可以不被另一个没有连接到 SetVariable 的角色所读取，SetVariable 可能导致模型的非确定性。即模型执行的结果可能取决于调度决策，它决定了 SetVariable 是否在角色读取受影响的变量之前执行。为了降低该风险，SetVariable 有一个称为 delayed（延迟）的参数，在默认情况下它设置为真。当它为真时，受影响的变量只会设置在指示器当前迭代的末端。对于大多数指示器（值得注意的是，除了 PN 或会话外），这个设置将保证确定性行为。SetVariable 角色有一个输出端口，它可以保证其他指定角色能在 SetVariable 执行后才开始执行。即使 delayed（延迟）设置成假，这种方法也能消除不确定性。

在图 2-11 所示的模型中，关系（Relation）用于将单个端口的输出信号广播到其他角色。这个单端口处仍仅一条连接——连接到关系。

即使目前建立的是一个简单的图来说，但布置图标以及控制端口和关系之间的连线，依然是很乏味的。幸运的是，PtolemyII 包含了一个智能的自动布局工具，你可以在菜单命令 [Graph → Automatic Layout] 中找到它（或者 Ctrl+T 调用）。该工具是由 Spönemann et al. (2009) 提供。然而，偶尔也需要手动调整布局。为了具体地控制连接的路径，你可以沿着一个连接使用多重关系，并且独立地控制每一个连接的位置。用于单个连接的多重关系称为关系组（relation group）。在关系组中，关系的连接顺序是没有意义的。

Ptolemy II 支持两种类型的端口，采用实心 and 空心三角形来表示。实心三角形指定单输入（或单输出）端口，而空心三角形允许多个输入（或输出）。例如，Ramp 角色的输出端是一个单端口（single port），而 Display 和 SequencePlotter 角色的输入端是多端口（multiport）。在多端口中，每个连接都被视作一个单独的信道（channel）。

从库中选择另一个信号源并将它添加到模型中。连接这个新增的源到 SequencePlotter 或者 Display，从而实现到这些块的多端口输入。然而，并不是所有数据类型的输入都可以被接受。比如 SequencePlotter 只接受 double 类型或者能无损地转换成 double 类型的输入，如 int 类型。在下一节中，将讨论数据类型以及它们在 Ptolemy II 模型中的使用。

2.2 令牌和数据类型

在图 2-7 的例子中, Const 角色在它的输出端口创建一个值序列。序列中的每个值称为一个令牌 (token)。令牌由角色创建, 然后通过输出端口传送出去, 最后被一个或多个目标角色 (destination actor) 接收。每个目标角色 (destination actor) 通过输入端口接收令牌。在本例中, Display 角色将接收由 Const 角色创建的令牌并将它们显示在窗口中。一个令牌就是一个数据单元, 就像一个字符串或者数值, 令牌在两个角色之间通过端口进行通信。

一个由 Const 角色创建的令牌, 它的值可以是 Ptolemy II 表达式语言 (见第 13 章) 能表达的所有值。可将该值设为 1 (整数 1), 或者 1.0 (浮点数 1), 或者 {1.0} (包含 1.0 的单元元素数组), 或 {value = 1, name = "one"} (一条由两个元素组成的记录: 一个名为 value 的整数和一个名为 name 的字符串), 或者 [1,0;0,1] (一个 2×2 单位矩阵)。这些都是 Const 角色的有效表达式。

Const 角色可以产生不同类型 (type) 的数据, Display 角色可以显示不同类型的数据。角色库中的大多数角色是多态角色 (polymorphic actor), 也就是说它们可以对多种类型的数据进行操作, 或者产生多种类型的数据, 即便它们的具体处理行为可能会因类型的不同而不同。比如, 矩阵乘法与整数乘法的操作并不相同, 但是它们都可以使用 Math 库中的 MultiplyDivide 角色来完成。Ptolemy II 包含了一个很复杂的类型系统, 它允许角色有效而安全地处理不同的数据类型。该类型系统是由 Xiong (2002) 创建的, 这个系统将在第 14 章中描述。

为了更进一步对数据类型的探讨, 我们使用 SDF Director 创建如图 2-12 中所示的模型。Ramp 角色的位置在 Sources \rightarrow SequenceSources 子库中, AddSubtract 角色在 Math 库中 (见 2.2 节的补充阅读)。将 Const 的 value (值) 参数设为 0, 指示器的迭代 (iteration) 参数设为 5。运行此模型, 结果会显示从 0 ~ 4 的 5 个连续数, 如图 2-12 所示。这些值是由 Ramp 的当前值减去 Const 角色的常数输出而产生的。请读者自己做实验, 改变 Const 角色的值, 观察输出端的 5 个数是如何变化的。

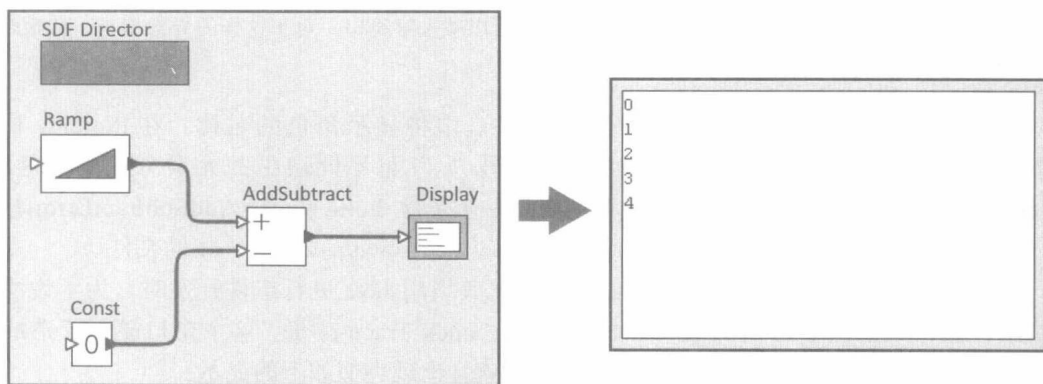


图 2-12 另一个例子, 用以解释数据类型

现在将 Const 角色的值改回 “Hello World”。当你执行这个模型时, 你应该会看到图 2-13 所示的异常 (exception) 窗口。出现这个错误是因为试图将一个整数值减去一个字符串值。这是一种类型错误 (type error)。

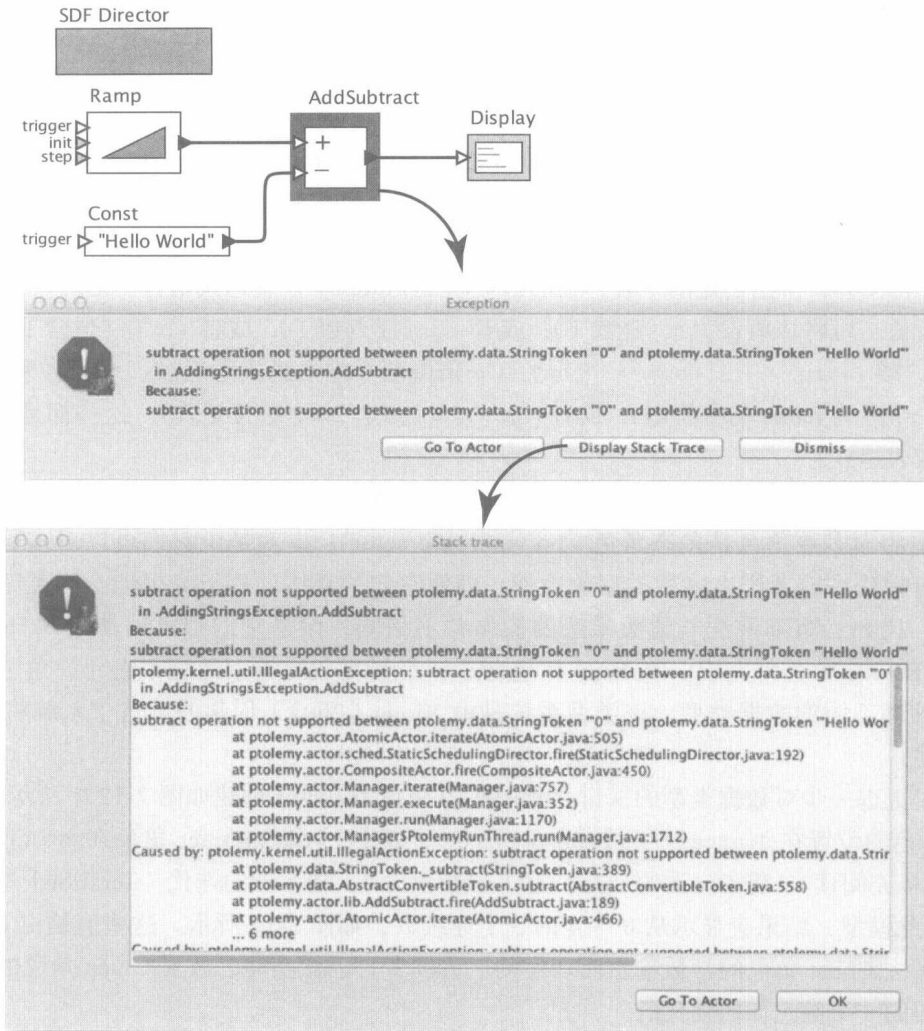


图 2-13 当试图执行该例时引起了一个异常。整数不可以减去字符串。对于这样的异常检查，栈跟踪是非常有用的，尤其在包含自定义角色时

引起异常的角色被高亮显示，并且异常窗口中将显示角色的名称。在 Ptolemy II 模型中，所有对象都有一个加“.”的名称。句号“.”将不同层中的元素分隔开。这样，“.helloWorldSubtractError.AddSubtract”就表示一个包含 .hello World AddSubtractError 错误的“AddSubtract”对象。（此模型保存为名为 helloWorldAddSubtract.xml 的文件。）

异常（Exceptions）是很有用的调试工具，尤其当用 Java 进行组件开发时。为了说明如何使用它们，单击图 2-13 异常窗口中的 Display Stack Trace 按钮。这个窗口显示了造成异常的执行序列。比如说，如果你向下滚动窗口，就会出现类似如下的显示：

```
at ptolemy.data.StringToken._subtract(StringToken.java:359)
```

在 String Token. Java 源代码中的 359 行出现异常是由 ptolemy.data.StringToken，类中的 subtract 方法引起的。因为 Ptolemy II 是和源代码一起发行的（大多数安装机制都支持源代码安装），因此这是非常有用的信息。对于类型错误，也许不需要查看栈追踪，但是如果

想用自己的 Java 代码扩展该系统或者遇到了一个无法解释的问题，那么查看栈追踪会得到启发。

为了寻找上面提到的 StringToken.java 文件，请查找 Ptolemy II 安装目录。如果目录是 \$PTII，那么文件的位置由完整的类名给出，但是点“.”换成了斜杠“/”，这种情况下，它在：

```
$PTII/ptolemy/data/StringToken.java
```

在 Windows 系统中这些“/”会变为“\”“/”。

因此可以尝试对模型进行修改来排除异常。将 Const 角色从 AddSubtract 角色的端（-）口断开，并将它连接到端口（+），如图 2-14 所示。步骤为：1）选中连接；2）删除（用 delete 键）；3）增加一条新连接或者选中它并从它的一个端点拖曳到另一个新位置。注意，高端口是空心三角形，这表明你可以对它建立一个以上的连接。当模型执行时，得到如图 2-14 所示的从“0Hello World”起的一个字符串序列。按照 Java 的惯例，字符串可以进行整数加（使用级联），但不能进行整数减。

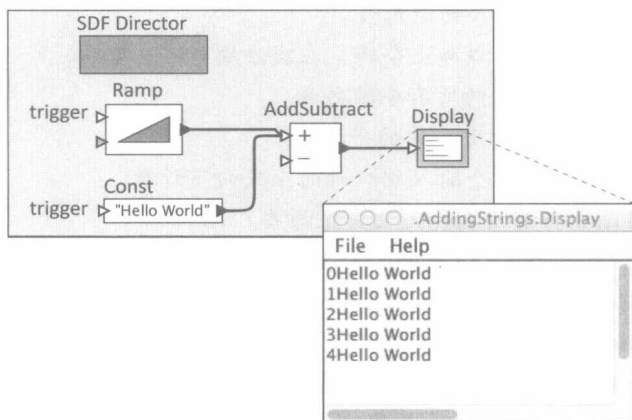


图 2-14 整数加字符串

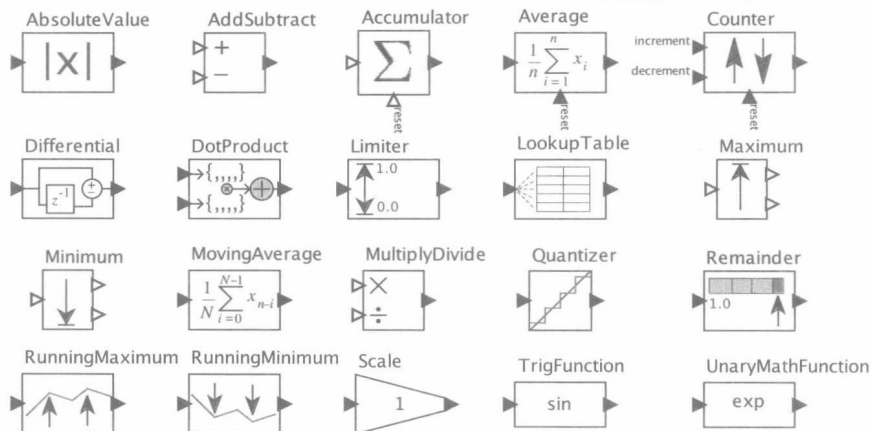
所有连接到一个多重端口的连接必须具有相同的类型。在这种情况下，多重端口的输入为一个整数序列（来自 Ramp）和一个字符串序列（来自 Const）。Ramp 整数自动转换为字符串，且与“Hello World”进行级联，以生成输出序列。

如上例所示，当角色需要且类型转换可行时，Ptolemy II 自动进行类型转换。通常，Ptolemy II 将在信息没有丢失的情况下自动进行类型转换。比如说，int 可以转换为 string，反之则不行。int 可以转换成 double，反之不行。int 可以转换成 long，反之则不行。在第 14 章将详细解释，但通常读者无须记住转换法则。一般情况下，数据类型转换和结果计算会如所期望的那样进行。

为进一步探索数据类型，可通过修改 Ramp 角色，使其参数为不同的类型。比如，试着将 init 型变成 step 型字符串。

补充阅读：Math 库

Actors → Math 库中的角色进行数学运算，如下图所示（除了最多功能的表达式外，其他的将在第 13 中解释）。



它们的意义大多可从名称上来直观理解:

- **AbsoluteValue** 计算输入的绝对值。
- **AddSubtract** 在加法端口加上令牌, 在减法端口减去令牌。
- **Accumulator** 输出到达的所有令牌之和。
- **Average** 输出到达的所有令牌的平均值。
- **Counter** 当令牌到达两个输入端口时向上或向下计数。
- **Differential** 输出当前输入和上次输入的差。
- **DotProduct** 计算两个数组或矩阵输入的内积。
- **Limiter** 将输入的值限制在某一范围内。
- **LookupTable** 根据输入查找表中的数组作为输出。
- **Maximum** 输出当前有效输入令牌的最大值。
- **Minimum** 输出当前有效输入令牌的最小值。
- **MovingAverage** 输出最近输入的平均值。
- **MultiplyDivide** 对乘法输入端的输入做乘法, 对除法输入端的输入做除法。
- **Quantizer** 从指定列表中选出最接近输入值的值进行输出。
- **Remainder** 将进行输入值除以 divisor 参数后的余数输出。
- **RunningMaximum** 将目前所有输入中的最大值进行输出。
- **RunningMinimum** 将目前所有输入中的最小值进行输出。
- **Scale** 将输入乘以一个常量参数。
- **TrigFunction** 计算三角函数, 包括正弦、余弦、正切、反余弦、反正弦和反正切。
- **UnaryMathFunction** 能够执行一元变量的函数, 包括求幂、求对数、求符号函数、求平方值、求平方根。

还有一些细节需要注意。一些有多输入端口 (**AddSubtract**、**Counter** 和 **MultiplyDivide**) 的角色, 或者一些有多重输入 (**Maximum** 和 **Minimum**) 的角色, 并不要求所有的输入信道都有输入令牌。当这些角色点火时, 一旦输入令牌可用, 它们就开始执行。比如说, 若 **AddSubtract** 端口的 + (plus) 信道上没有任何令牌, 只有 - (minus) 信道上有一个令牌, 那么输出将是这个令牌取负所得的值。当角色点火时, 输入是否可用取决于指示器和模型本身。若是 SDF 指示器, 每次点火时, 每个输入信道上只严格地提供一个输入令牌。

这些角色是**多态的** (polymorphic)，它们可以作用于多种不同的数据类型。比如说，AbsoluteValue 角色接收任何标量类型的输入 (见第 14 章)。如果输入类型是复数，则它计算模值。对于其他标量类型，它计算绝对值。

Accumulator 和 Average 都有复位 (reset) 输入端口 (在图标底部)。Accumulator 和 Average 计算输入序列的和或者均值，且它们都可以被复位。

2.3 层次结构和复合角色

Ptolemy II 支持 (和鼓励) 层次化模型。这种模型包含了本身就是模型的组件。这些组件称为复合角色 (composite actor)。

考虑一个从噪声通道中恢复一个信号的典型信号处理问题。使用 Ptolemy II，创建一个复合角色进行一个有噪声通信通道的建模，并在更大的模型中使用该角色。

为了创建复合角色，从 Utilities 库中拖出一个 CompositeActor。如图 2-15 所示，在上下文菜单 (通过右击复合角色获得) 中，选择 [Customize → Rename] 给该复合角色一个适当的名称，如 Channel。(注意，你也可以提供一个 Display name (显示名称) 参数，它可以是任意文本，它将替代原角色名的显示内容。) 然后，再使用上下文菜单，选择 Open Actor。如图 2-16 所示，它将打开一个新的图形编辑器窗口。注意，原始的图形编辑器依然是打开的。可通过拖动新窗口的标题栏来移动新的图形编辑窗口，将其显示。也可以单击工具栏中向上指向的三角形返回原始图形编辑器。

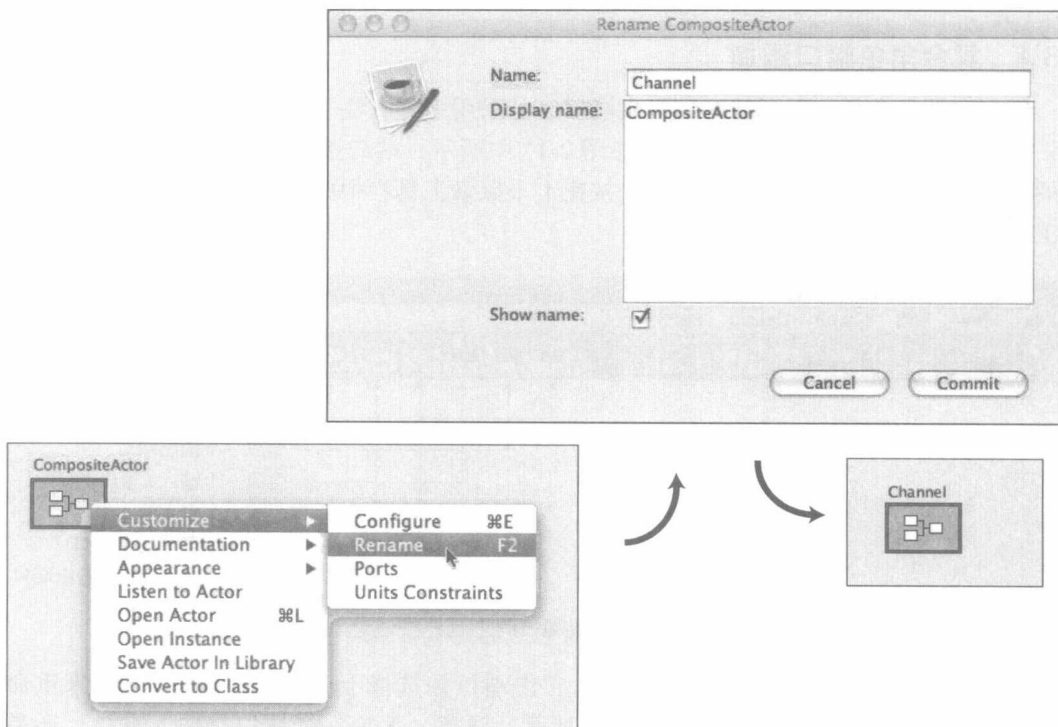


图 2-15 重命名角色

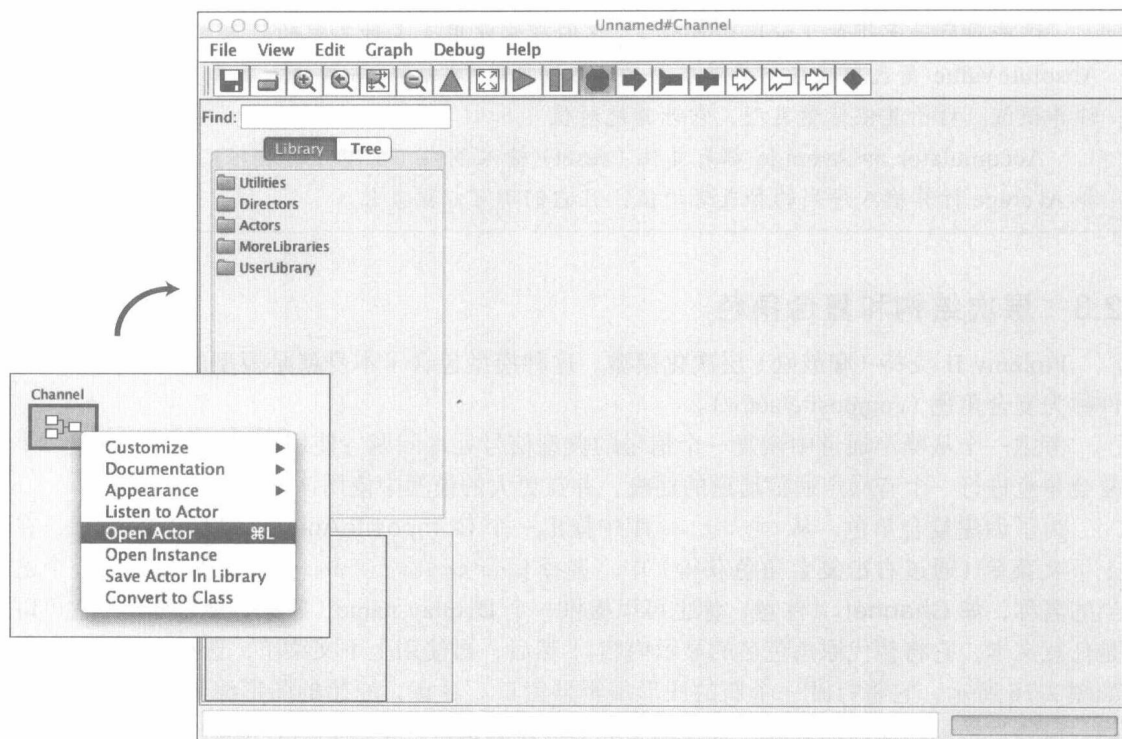


图 2-16 打开一个新的复合角色，它显示了空的角色内部

2.3.1 复合角色端口添加

新创建的复合角色需要输入和输出端口。可采用多种方法来添加输入输出端口，最简单的一种是单击工具栏上的端口按钮。如图 2-17 中所示，端口按钮显示为工具栏中白色或黑色的箭头。可以通过将鼠标停留在每个按钮上来探索工具栏中的按钮，将弹出描述这些按钮的工具提示。

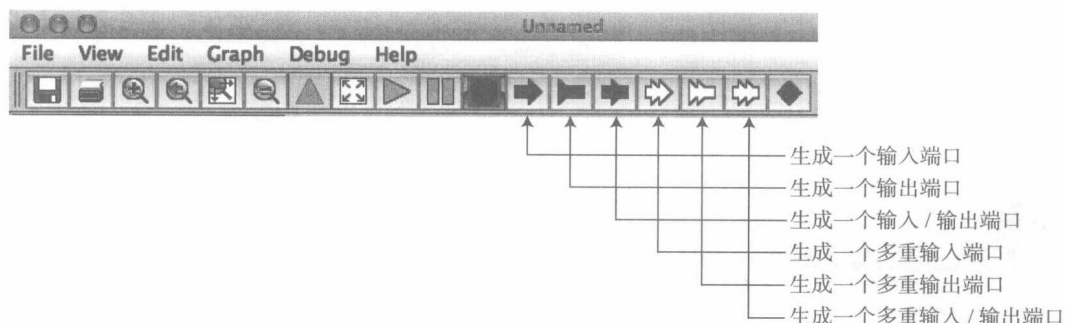


图 2-17 创建新端口的工具栏按钮总结

创建一个输入端口和输出端口，通过右击端口并选择 [Customize → Rename] 来重命名 input (输入) 和 qput (输出)。如图 2-18 所示，注意，无论一个端口是输入端口，还是输出端口或是多端口都可以右击复合角色的背景并选择 [Customize → Ports] 来添加、移除或改变。弹出的对话框也可进行端口类型的设置，即使通常你不需要设置端口类型，因为

Ptolemy II 从端口连接 (connection) 中决定端口类型。也可以指定端口的方向^①并且指定是否在图标外部指定端口名字 (默认为否), 或者端口是否显示。

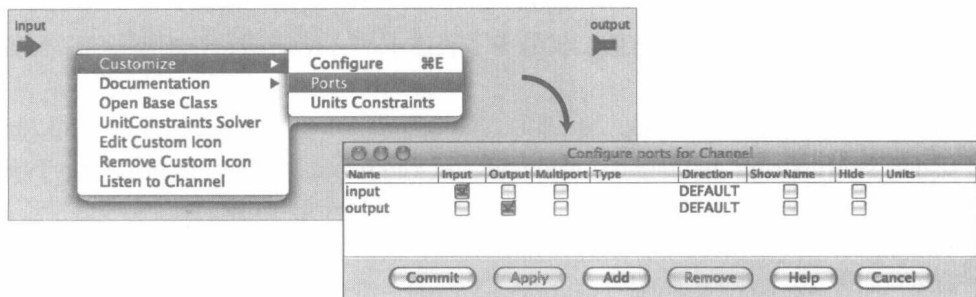


图 2-18 右击背景会出现一个对话框, 它可以用来配置端口

使用这些端口, 创建如图 2-19 所示的模型^②。利用在 Random 库中的高斯 (Gaussian) 角色, 创建服从高斯分布的随机变量。如果返回上层模型, 能够很轻松地创建如图 2-20 所示模型。Sinewave 角色在 Sources → SequenceSources 下, SequencePlotter 角色在 Sinks → SequenceSinks 下。Sinewave 角色也是一个复合角色 (可以打开该角色)。如果执行该模型 (可以给 interaction 参数设置一个合理的值, 如 100), 可以看到类似图 2-20 所示的图。

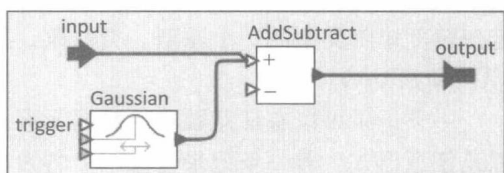


图 2-19 一个简单的定义为复合角色的信道模型

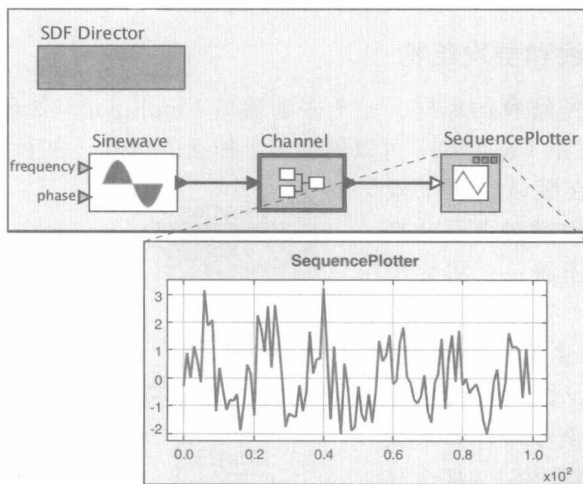


图 2-20 一个简单的信号处理例子, 在正弦信号中加入噪声

2.3.2 端口类型设置

在上述例子中, 无需定义端口类型。它们的类型可以从模型中的连接 (connection) 和常量中推断出, 该问题将在第 14 章中解释。然而, 偶尔也需要进行端口类型的设置。注意

① 一个端口的方向由它出现在角色图标上的位置决定。默认情况下, 输入端口在左, 输出端口在右, 既是输入又是输出的端口在图标底部。

② 提示: 若要在对外端口上创建一个连接, 请在拖曳时按下 Control 键或在 Mac 系统上按 Command 键。

图 2-18 中的对话框有一列可进行端口类型的指定。比如，若要指定端口为布尔类型，你可以输入 `boolean` 即可。但是，仅当端口属于不透明的复合角色（有自己的指示器）时，这样做才有效。但在已经建立的模型中，`Channel` 复合角色是透明的，所以对它的端口类型进行设置将没有实际作用。透明复合角色仅仅是为了表述上的方便简洁，在模型执行中它们的复合端口不起实际作用。

通常使用的类型包括 `complex`、`double`、`fixedpoint`、`float`、`general`、`int`、`long`、`matrix`、`object`、`scalar`、`short`、`string`、`unknown`、`unsignedByte`、`xmlToken`、`arrayType(int)`、`arrayType(int, 5)`、`[double]` 以及 `{x=double, y=double}`。类型的详细描述在第 14 章。

在 Ptolemy II 的表达语言中，矩阵使用方括号，数组使用大括号。数组是任意类型令牌的有列表。一个 Ptolemy II 矩阵是包含数值类型的一维或二维的结构。例如，若指定端口为双精度矩阵类型，使用下面的表达式：

```
[double]
```

这个表达式建立了一个 1×1 的矩阵，它包含了一个 `double` 类型（具体值暂不讨论）。它是指定双精度矩阵类型的一个原型。类似地，我们可以指定一个复数类型的数组为：

```
{complex}
```

一条记录可以有任意数量的数据元素，它们中的每一个都有自己的名字和值，其中值可以是任意形式。若一条记录包含一个名为“name”的字符串和一个名为“address”的整数，可使用下面的表达式来指定：

```
{name=string, address=int}
```

关于数组、矩阵和记录的细节将在第 13 章给出。

2.3.3 多端口、总线和层次结构

如上面的 2.1.3 节所解释的那样，一个多重端口（multiport）处理多个独立的信道。在类似的方式中，一个关系（relation）可以处理多个独立的信道。一个关系（relation）有一个 `width`（宽度）参数，在默认情况下设置为 `Auto`，意味着可以通过上下文来推断宽度。如果宽度不统一，将会出现图 2-21 所示的情况：关系（relation）上的一条斜线旁边标注了一个数字。这样的连接称为**总线**，因为它装载多个信号。在图 2-21 的例子中，如果设置的复合角色的关系宽度为 2，那么只使用了复合角色的 $2/3$ 的信道。

在大多数情况下，关系的宽度可从具体使用中推断（Rodiers and Lickly, 2010），但偶尔显式地设置它也是很有必要的。通过使用 `BusAssembler` 角色也可以显式地构造一个总线，或者使用 `BusDisassembler` 角色从总线中分离部分信道，两个角色都可以在 `FlowControl` → `Aggregators` 库中找到。

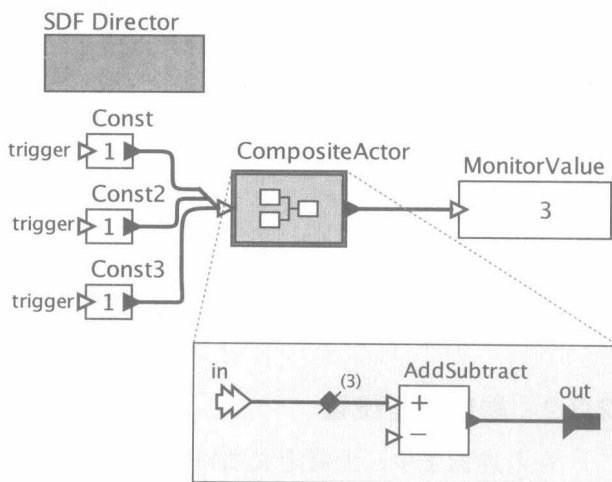


图 2-21 关系的宽度可以大于 1，这意味着它们携带了多个信道。图中复合角色中的关系的宽度推断是 3 个

2.4 注释及参数设置

本节将从多个方面改进图 2-20 的模型。比如参数添加、修饰和注释文档的插入以及角色图标定制等。

2.4.1 层次化模型中的参数

首先，注意图 2-20 中，噪声覆盖了正弦曲线，使其几乎不可见。对这个信道模型的一个有效的修改方法是：添加一个**参数 (parameter)**，用来设置噪声的强度。为了实现这种改变，右击信道角色打开信道模型，并选择 **Open Actor**。在信道模型中加入参数的方法是，在 **Utilities** → **Parameters** 子库中选择一个参数并将它拖进模型，如图 2-22 所示。右击参数将它重命名为 **noisePower**（为了在表达式中使用参数，在 **Parameter** 中其名字不能有空格）右击（或双击）参数，将它的值改为 0.1。

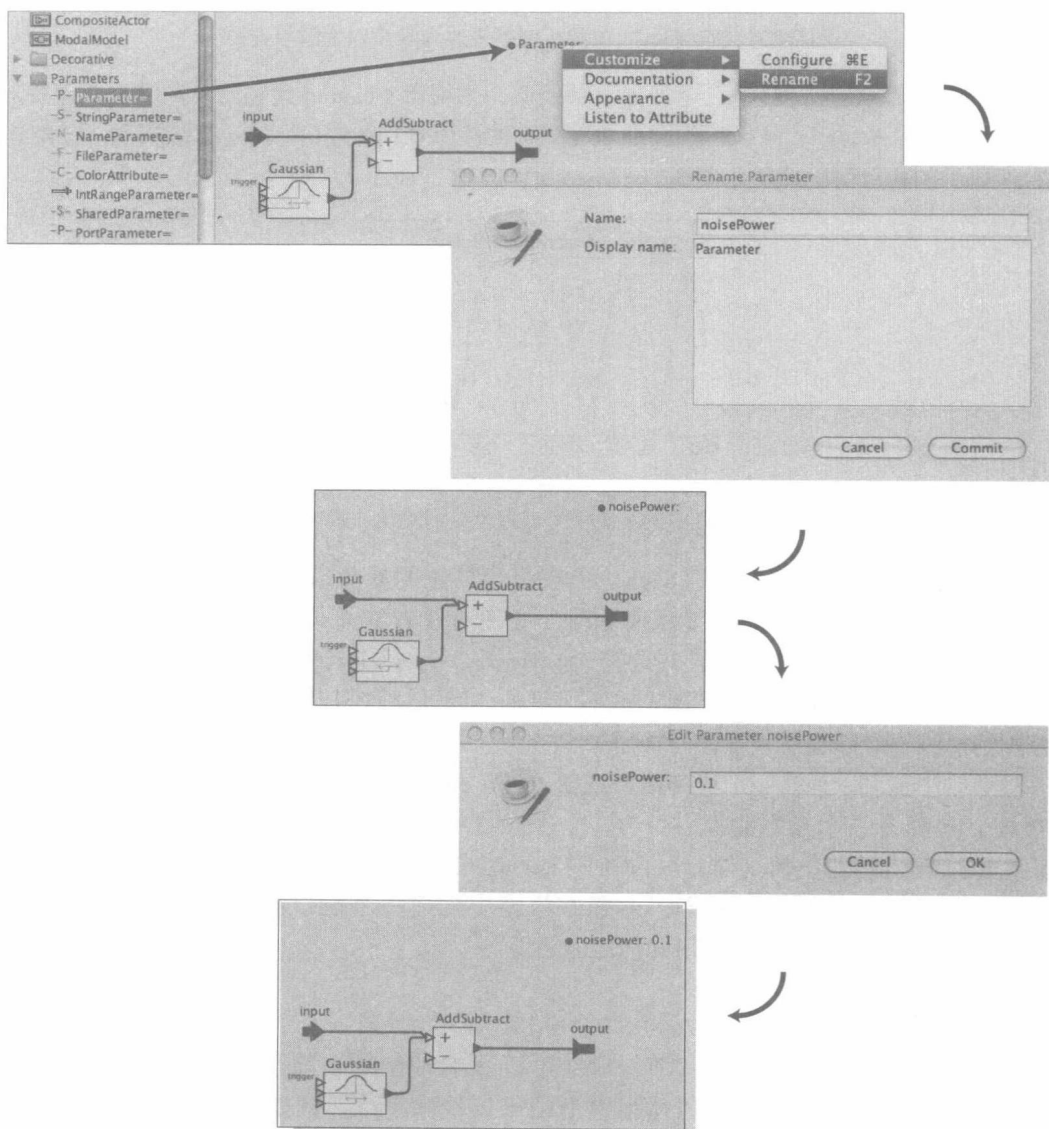


图 2-22 在信道模型中添加参数

现在，这个参数可以用于设置噪声的大小。Gaussian 角色（高斯角色）有一个称为 `standardDeviation`（标准差）的参数。因为噪声强度等于方差而不等于标准差，所以将 `standardDeviation` 的值改为 `sqrt(noisePower)`，如图 2-23 所示。（关于表达式语言，详见第 13 章）

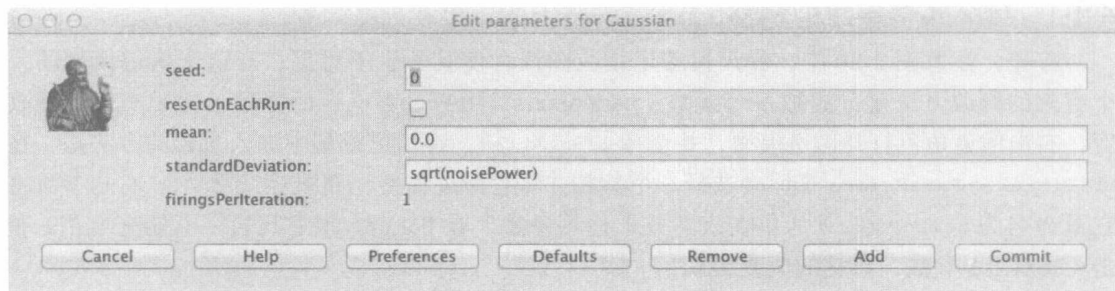


图 2-23 Gaussian 角色的标准差被改为噪声强度的平方根

为了观察参数造成的影响，返回到顶层模型，并编辑 Channel 角色的参数（双击或右击并选择 [Customize → Configure]）。将 `noisePower` 从默认值 0.1 改为 0.01。运行该模型。应能得到图 2-24 所示的相对清晰的正弦曲线。

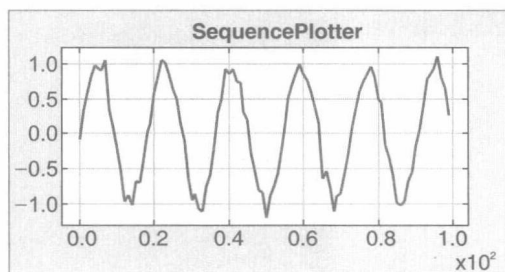


图 2-24 图 2-20 中的简单信号处理模型的输出，噪声强度 = 0.01

还可以为复合角色添加参数，方法是在复合角色的参数编辑对话框中单击 Add 按钮。该对话框可通过双击 Channel 图标来访问，或者通过右击并选择 [Customize → Configure]，或者通过单击复合角色内的背景并选择 [Customize → Configure]。这里需要注意的一个关键问题是，这种方式添加的参数在图中是不可见的，因此这种机制应该谨慎使用，并且仅当有充分理由对浏览者隐藏模型参数时使用。

最后，注意可以创建一个称为端口参数（port parameter）或者参数端口（parameterport）的对象，它既是一个参数也是一个端口。图 2-5 中的 `frequency` 和 `phase` 对象就是参数端口。它们像任何其他参数一样，可以在一个表达式中访问，但当有一个输入在参数端口执行期间到达时，其参数值会被更新。若想在模型中创建一个端口参数对象，只需简单地从 Utilities → Parameters 库中拖出并为之命名即可。

2.4.2 修饰元素

也可以使用各种修饰元素（即，那些可以影响模型外表而不影响其功能的元素）来进行模型的修饰。这些元素可以增强模型的可读性和美感。例如，试着从 “Utilities → Decorative” 子库拖拽一个 Annotation（注释）图表，并命名。这种注释非常值得推荐，它

们相当于程序中的注释，可以极大地提高程序的可读性。其他修饰元素（如几何图形）可以从相同的库中拖拽。

2.4.3 创建自定义图标

Vergil 提供了图标编辑器，它允许用户创建自定义角色图标。创建自定义角色图标的方法是：右击标准图标并选择 [Appearance→Edit Custom Icon]，如图 2-25 所示。图标编辑器中间的方框显示默认图标的尺寸，以供参考。创建一个如图 2-26 所示的图标。提示：矩形的填充颜色设置为 none，设置四边形的填充颜色时首先使用颜色选择器，然后再调整透明度 alpha 为 0.5。最后，因为图标本身带有角色名，打开 [Customize→Rename] 对话框可选择显示角色名。

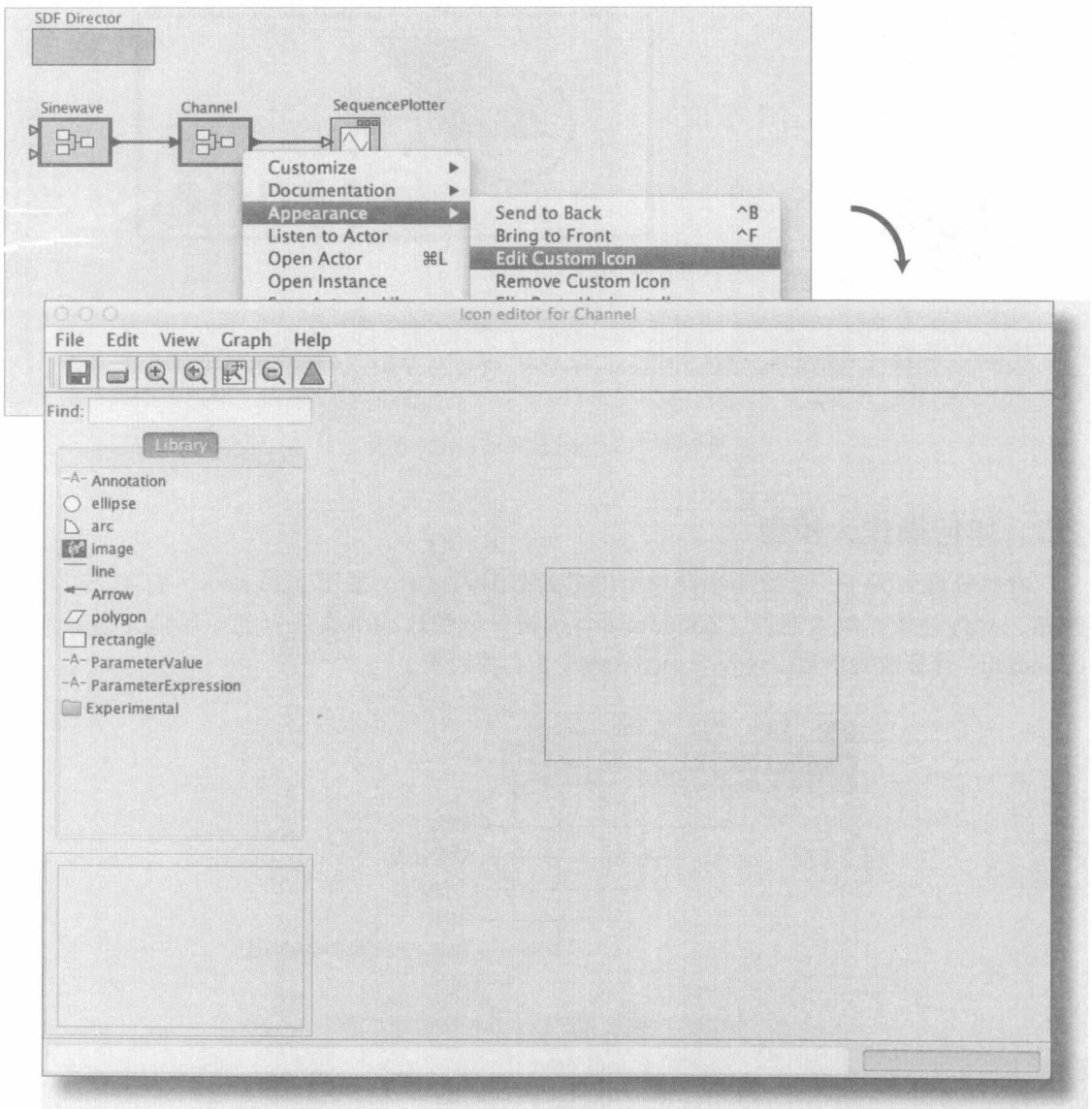


图 2-25 Channel 角色的自定义图标编辑器

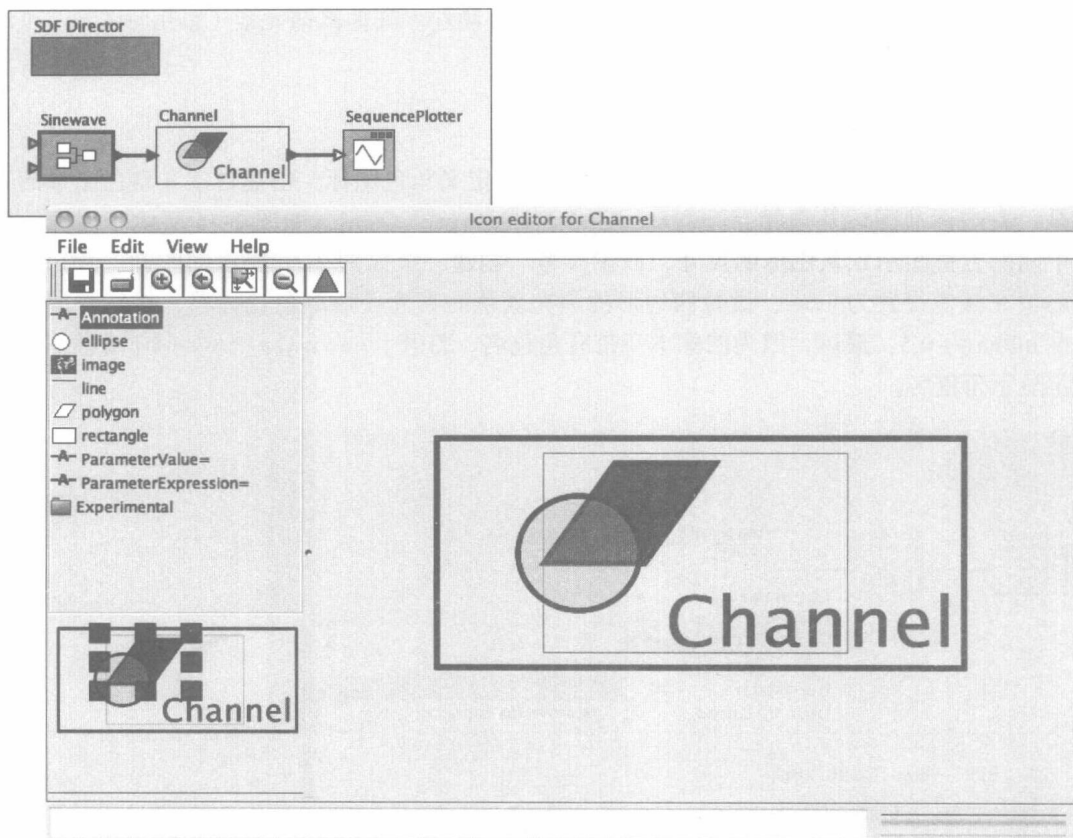


图 2-26 Channel 角色的自定义图标

2.5 如何操作大模型

有些模型太大了，以致无法在单个屏幕的范围内查看。如图 2-27 所示，有 4 个工具栏按钮，允许缩放。放大还原（Zoom reset）按钮可以将放大倍数恢复到原始值，合适尺寸（Zoom fit）计算放大倍数以便整个模型在编辑窗口中可见。

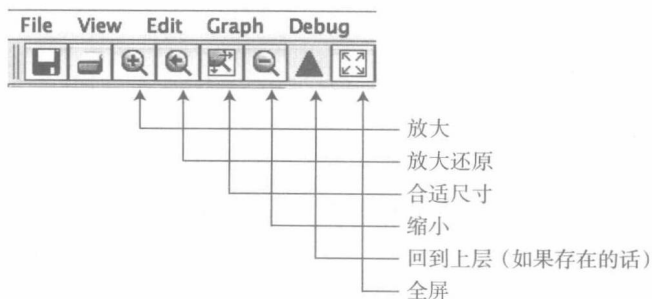


图 2-27 缩放和调整工具栏按键的总结

也可以导航整个模型。如图 2-28 所示的窗口。放大图标使得图标比默认大小更大。左下角的导航窗口（pan window）显示整个模型，用红色方框表示模型在屏幕上的可见部分。通过单击或拖动导航窗口，可以很容易地对整个模型进行操作。

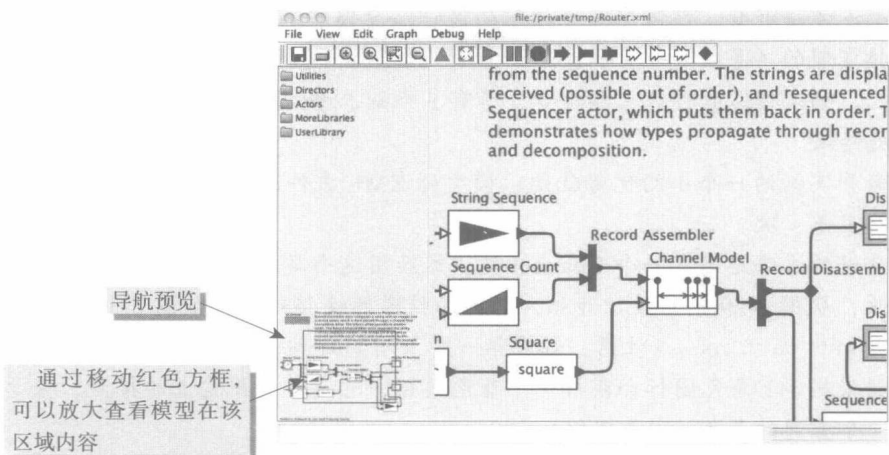


图 2-28 左下角的导航预览中的红色框代表在主编辑器窗口中模型的可视区域。可通过移动红框来查看模型不同部分细节

2.6 类和继承

Ptolemy II 可以定义面向角色的类 (actor-oriented class)(Lee et al., 2009a)。这些类被定义后可以用于实例 (instance) 和子类 (subclass) 的创建, 它们都使用了类继承 (inheritance) 的概念。类为角色提供通用的定义 (或模板)。实例是类的一个单独的、特定的实现, 一个子类源于父类——它包含和父类相同的结构, 但可能有一些修改。这种方法提高了设计的模块化, 详见下面的例子。

例 2.1 结合在 2.3 节中开发的模型, 如图 2-29 所示。假设希望创建信道角色的多个实例, 如图 2-30 所示。在图 2-30 中, 正弦波信号通过 5 个不同的信道 (注意正弦波和信道之间的黑色菱形关系符号, 它表示将相同的信号分别广播到 5 个信道)。信道的输出被叠加在一起并以图形显示。结果是一条明显比单信道输出正弦波清晰的正弦波。(在通信系统中, 这种技术称为分集系统。信道的多个副本 (每个副本都携带独立噪声) 用于实现可靠通信。)

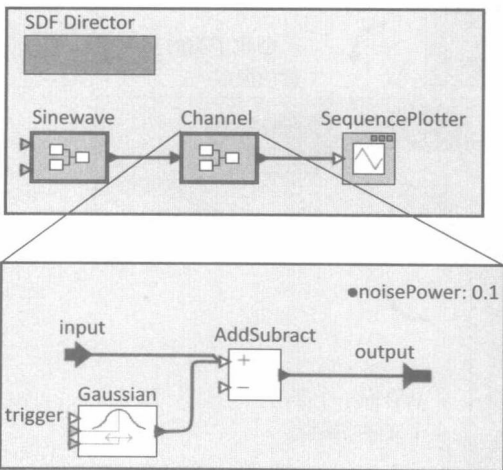


图 2-29 修改层次化模型以便使用类

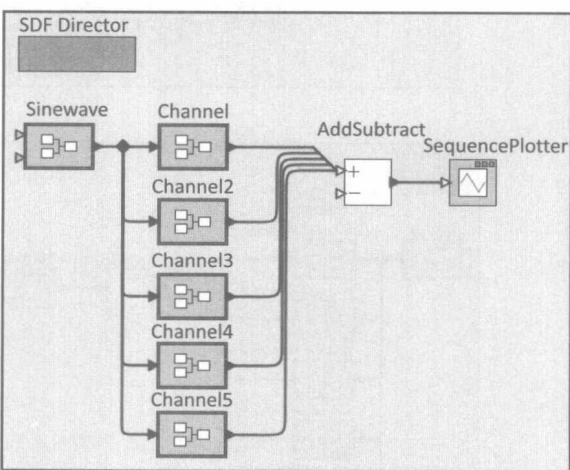


图 2-30 一个多样通信系统的简单设计, 该设计有多个如图 2-29 所示的信道副本

尽管上述方法有效，但依然是个糟糕的设计，原因有两点。首先在模型中，5个信道是通过硬连接实现的（将在下一节解释这个问题）。其次，每个信道都是图 2-29 中复合角色的副本。因此，如果信道的设计需要改变，所有 5 个副本都必须改变。因此这种方法使得模型难以维护或修改。

使用类和实例的一个小的优势就是，模型的 XML 文件表示会更小，因为类的设计只给出一次，而不是多次。

一个更好的方法是定义一个 Channel 类，然后用这个类的实例来实现分集系统。为了实现这种修改，从图 2-29 中的设计开始，将连向信道的连接删去，如图 2-31 所示。然后单击并选择 Convert to Class。（注意，如果第一步没有将那些连接删去，那么当进行试图转化时会出现错误提示）角色图标会获得一个蓝色边框，它是一个视觉上的提示，提示这是一个类而不是一个普通的角色（它是实例）。

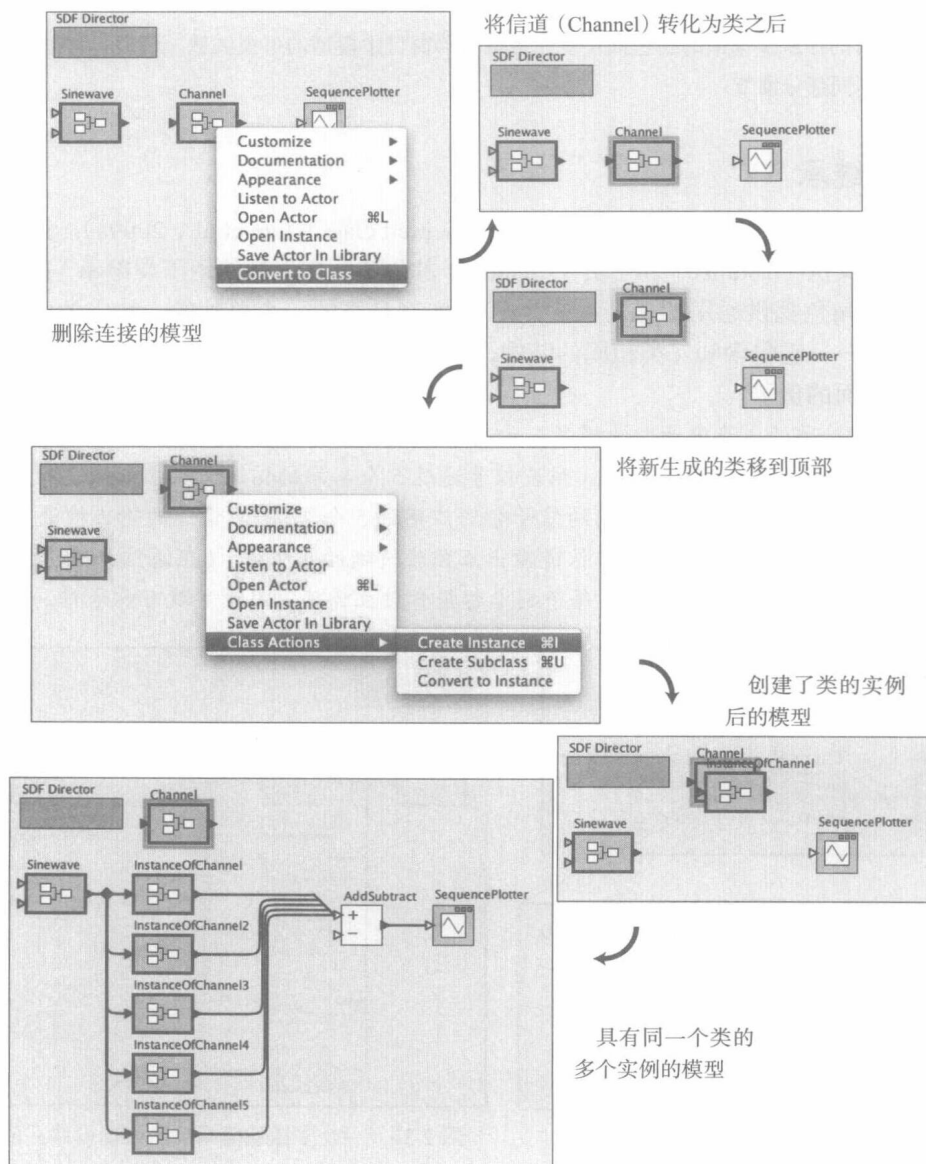


图 2-31 创建并使用一个信道 (Channel) 类

类在模型执行中不起任何作用，它只是必须被实例化组件的定义而已。按照惯例，将类置于模型的顶部，挨着指示器，因为它们都是表示声明。

还请注意，除了使用 `Convert to Class` 外，还可以从 `Utilities` 库里拖拽一个 `Composite ClassDefinition` 的实例，然后选择 `Open Actor` 并填入类定义。然后给这个类定义起一个具体的名字，比如 `Channel`。

当已经定义了一个类时，可以通过右击然后选择 `Create instance` 或者按 `Control+N` 键来创建一个实例。如此重复 5 次，创建 5 个实例，如图 2-31 所示。虽然这和图 2-30 的设计看起来很相似，但是实际上这是一个更好的设计，原因前文已经讲过。试着对这个类进行一些改变，比如说，为它创建一个自定义图标，如图 2-32 所示，可见类的变化将传播到类的每个实例。

如果在图 2-32 的任一实例（或类）上调用 `Open Actor`，就会看到相同的信通（`Channel`）模型。实际是类的定义。在这个层次化模型内所做的任何改变都会自动地传递到所有的实例。比如，用户可以试着修改 `noisePower` 参数的值并观察结果。

如果想查看实例而不是类定义，可以在一个实例上选择 `Open Instance`。打开的窗口只显示该实例。从类定义继承的每个子组件都由粉红色虚线框高亮显示。

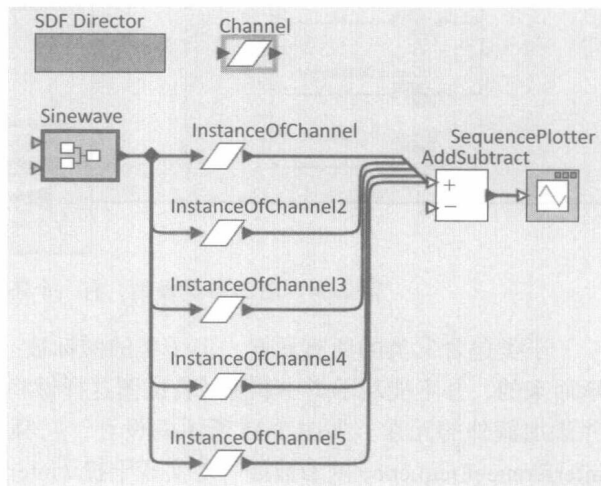


图 2-32 图 2-31 中的模型，类的图标变化了。基类的变化传播到每个实例

2.6.1 实例中参数值的重写

默认情况下，图 2-32 中的所有 `Channel` 类的实例都有相同的图标和相同的参数值。然而，每个实例都可以通过重写这些值进行定制。比如说，在图 2-33 中，对自定义图标进行了修改以获得不同的颜色，且第 5 个实例有了一个额外的图形元素。这些修改是由右击实例图标并选择 `Edit Custom Icon` 来实现的。若实例中类的参数进行了重写，那么更新类不会影响该实例，只会影响没有重写的实例。

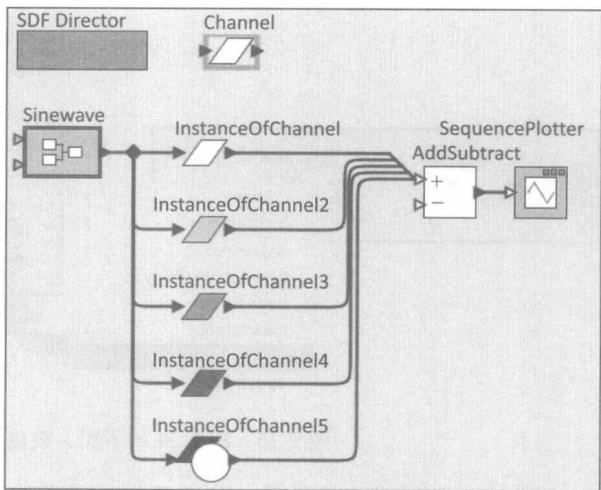


图 2-33 图 2-31 中的模型，实例的图标改变了，用于重写类中参数

2.6.2 子类 and 继承

假设需要对一些信道进行修改，用另一种形式的正弦波向其添加干扰。一种不错的方法是创建 `Channel` 类的一个子类，如图 2-34 所示。创建方法是右击类的图标，并选择 `Create`

Subclass。产生的子类图标出现在父类图标的上方，用户可自行移动。

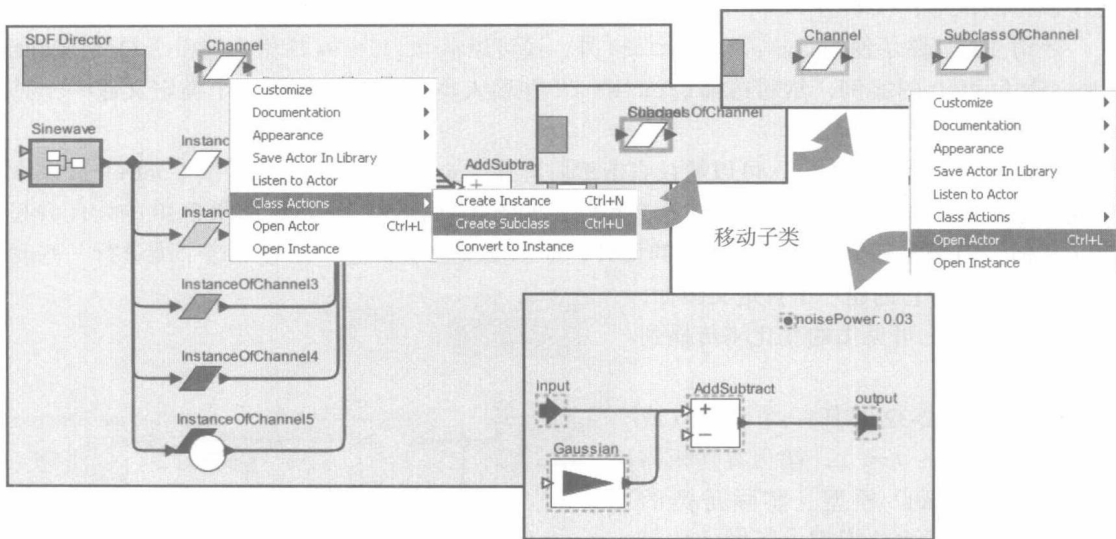


图 2-34 图 2-33 的模型，有一个还未被重写的 Channel 的子类

子类包含父类的所有元素，但子类的图标被一个粉红色的虚线框环绕。这些元素都是继承而来的，并不能从子类中删除（若试图这样做将出现错误提示）。但是，你可以改变参数值并添加额外的元素。如图 2-35 所示的设计，它额外增加了一对名为 *interferenceAmplitude* 和 *interferenceFrequency* 的参数如一对实现干扰（*interference*）的角色。

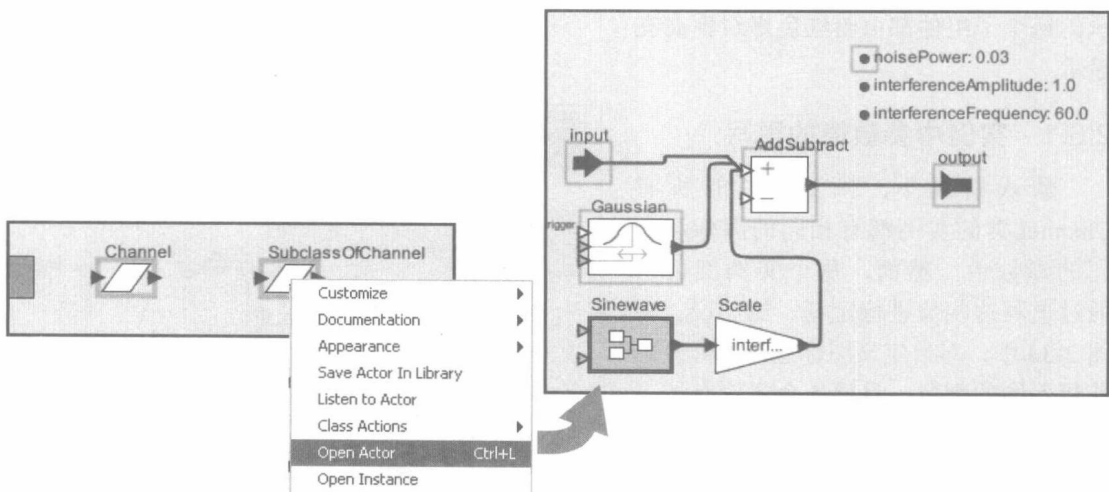


图 2-35 图 2-34 的子类，被重写过，加入了正弦干扰

图 2-36 中的模型将最后的信道替换为子类的一个实例，并用图显示正弦干扰。

类的实例有两种位置：第一，类的实例位于这个类本身所在的复合角色中；第二，类的实例位于一个复合角色中，且这个复合角色本身以及类本身都包含在一个模型中（就是说，类的实例位于子模型中）。为子模型添加一个实例，只需简单地从包含这个类的复合模型中复制（或剪切），然后将它粘贴到子模型中。

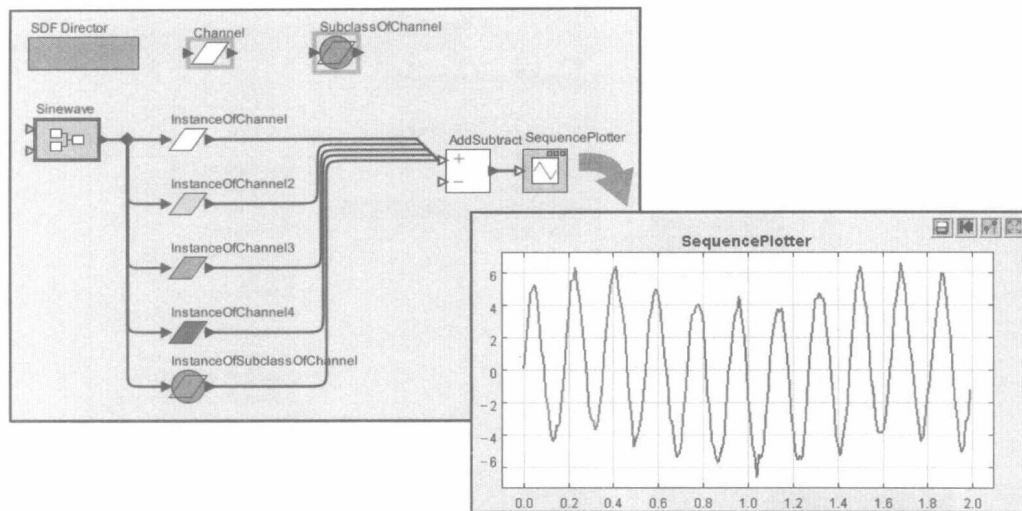


图 2-36 使用图 2-35 中子类的模型和执行图

2.6.3 模型间类的共享

通过将类的定义保存在独立文件中，一个类可以被多个模型共享。我们将用 Channel 类来说明这种技术。首先，在 Channel 类上右击并调用 Open Actor，然后在 File 菜单上选择 Save As。出现如图 2-37 所示的对话框。标记为 Save submodel only 的方框是默认未选中的。如果复选框未被选中，那么整个模型会被保存^①。在我们的例子中，我们想只保存信道子模型，所以需要选中复选框。

把类定义保存在一个模型可以访问到的地方是很重要的。一般地，Ptolemy II 搜索与类路径（classpath）相关的类定义，类路径由一个名为 CLASSPATH 的环境变量给出。原则上，你可以设置这个环境变量使它包括你想要搜索的任何目录。然而，在实际中，改变 CLASSPATH 可能会导致其他程序出问题，所以我们建议将文件存储在 PtolemyII 安装目录或根目录下名为“.ptolemyII”的目录中^②。在这两种情况下，Ptolemy 会发现存储在这些目录中的类文件。

如果将 Channel 类保存在名为 Channel.xml 且位于目录 \$PTII/myActors 下的文件中，这里 \$PTII 表示 Ptolemy II 的安装位置。该类定义可以用在任何模型里，如下方式：打开模型并在 Graph 菜单中选择 Instantiate Entity，如图 2-38 所示。在类路径中输入与 \$PTII 相关的类的全名，在本例中是 myActors.Channel。

当拥有一个定义在文件中的 Channel 类的实例时，可将其添加到 UserLibrary。UserLibrary 在 Vergil 库浏览窗口的左侧，如图 2-39 所示。右击实例并选择 Save Actor in Library。如图 2-39 所示，这使得另一个窗口被打开并显示用户库。用户库本身就是一个存储在 XML 文件中的 Ptolemy II 模型。当你保存库模型时，对于任何 Vergil 窗口（对于同一用户），类实例在 UserLibrary 窗口就变成可用的。注意，在用户库中保存类定义本身（相对于类的一个实例）结果会不同。在这种情况下，用户库将提供一个新的类定义而不是类的实例。

① 在某些平台上，会以单独的对话框显示 Save submodel only 复选框。

② 若你不知道 Ptolemy II 安装在系统的哪个路径下，你可以打开 [File->New->Expression Evaluator]，输入 PTII 并回车。或者，在图形编辑器中，选择 [View->JVM Properties]，然后查找 ptolemy.ptii.dir。

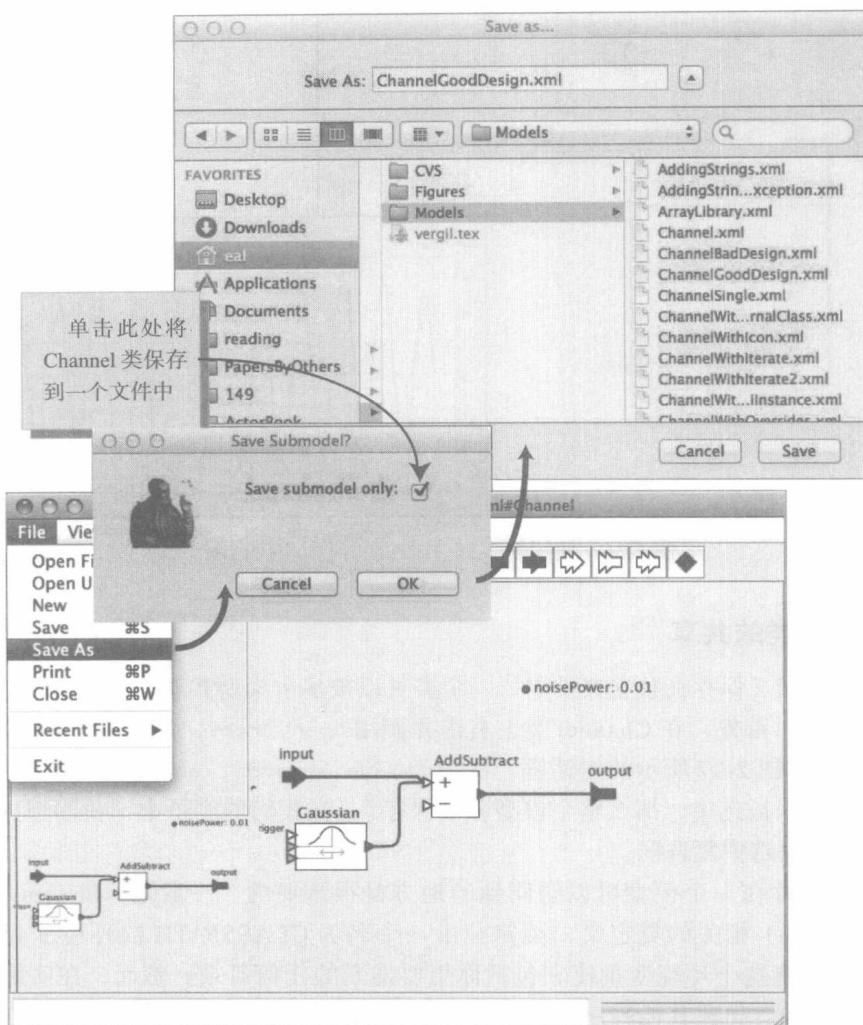


图 2-37 一个类可以单独保存在一个文件中，然后在多个模型中共享。如图所示，在某些平台上，显示有 Save submodel only 复选框的对话框会独立出现。在其他平台上，复选框被整合到一个整体对话框中。对于其他平台，复选框将集成在一个单独的对话框中

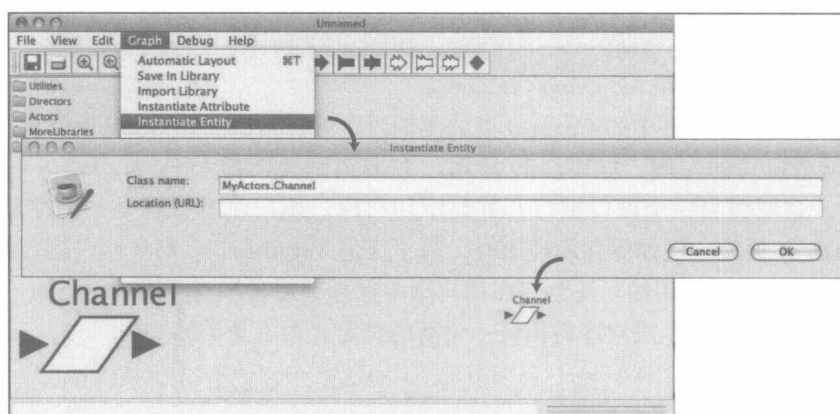


图 2-38 在文件中定义的类，可以在 Graph 菜单中使用 Instantiate Entity 来创建

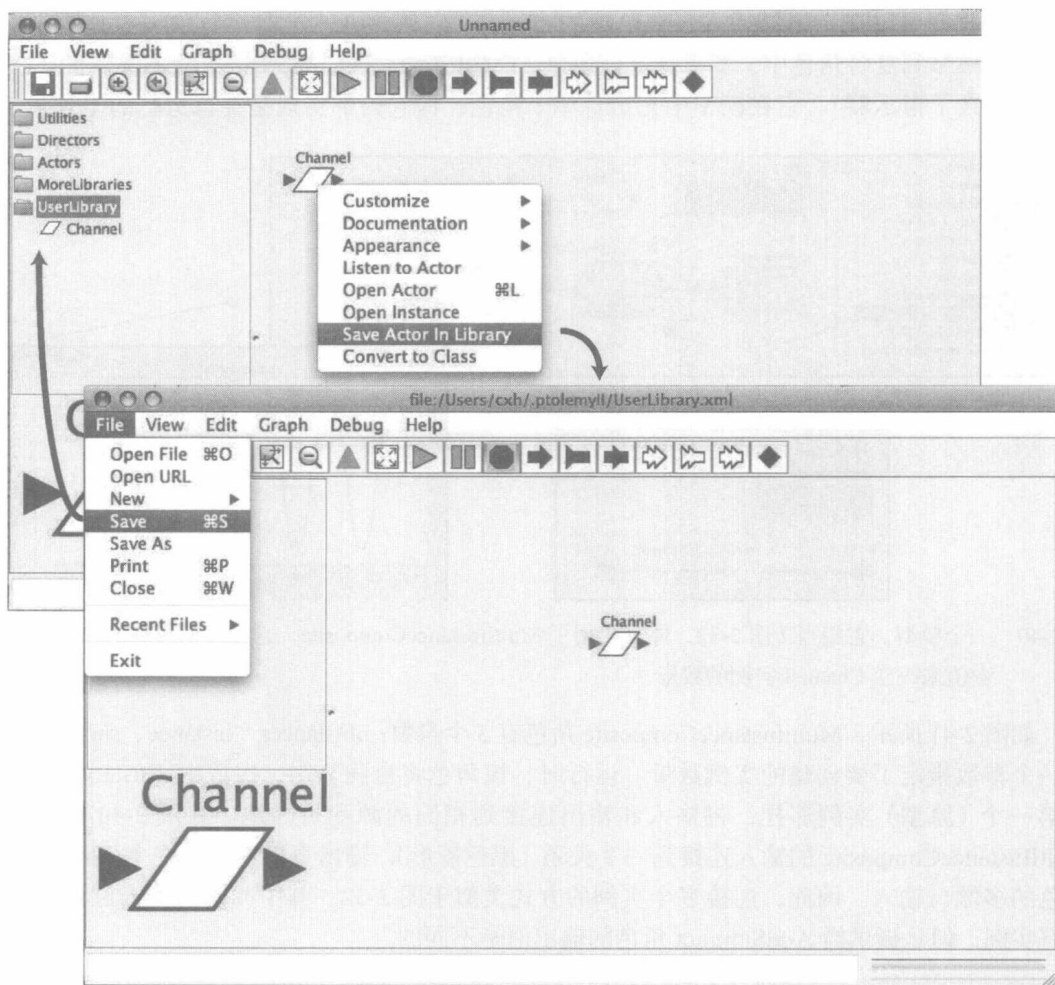


图 2-39 类的实例在自己的文件中定义，可以在 UserLibrary（用户库）中找到

2.7 高阶组件

Ptolemy II 包含大量的高阶组件（higher-order component）。这些角色对模型的结构进行操作，而不对输入数据进行操作。接下来将介绍几个相应的例子，在这些例子中，5 个 Channel 角色被置于一个模型中。为什么用 5 个？也许最好是有一个单独组件可以表示 Channel 的 n 个实例，其中 n 是一个变量，这正是高阶组件所能解决的问题。高阶角色由 Lee 和 Parks（1995）提出，它使得在模型结构不依赖于问题规模时更容易进行大型设计的构建。在本节中，将对这些角色进行描述，并且它们都能在 HigherOrderActors 库中找到。

2.7.1 MultilInstanceComposite 角色

考虑图 2-32 所示的模型，有 5 个并行连接的 Channel 类实例。实例的数量在图中是硬连接的，很难进行数量的改变，特别是如果需要增加它时。这个问题可通过使用 MultilInstance Composite（多实例复合）角色[⊖]来解决，如图 2-40 所示。MultiInstanceComposite 是一个复

[⊖] 由 Zoltan Kemenczy 和 Sean Simmons 提出，来自 RIM 有限公司（Research In Motion Limited）。

合角色，我们可以在它里面插入一个单独的 Channel 实例（通过创建一个 Channel 实例，将其复制并粘贴到复合角色中）多实例复合角色。（MultiInstanceComposite）是不透明的（这意味着它包含了指示器）。它在模型中充当着单个角色，但它内部实现是并行运行的多重角色。

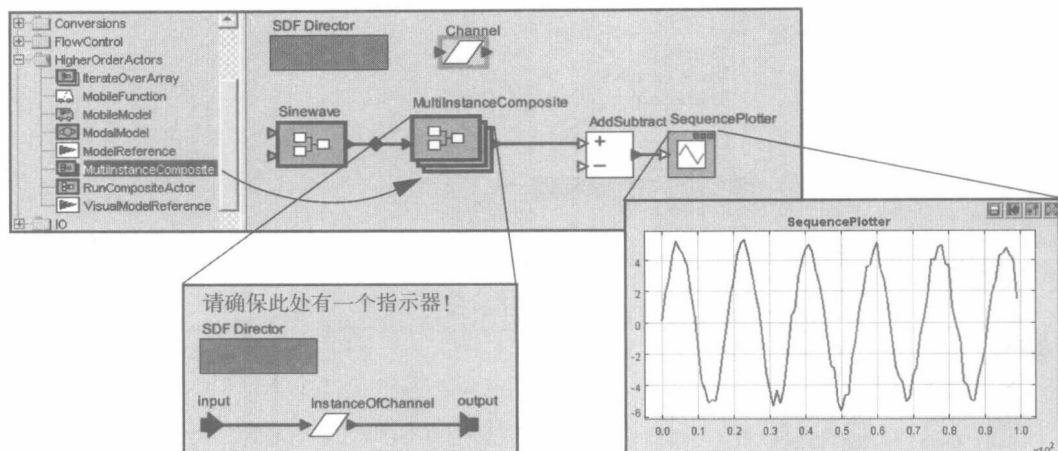


图 2-40 一个模型，它相当于图 2-32，但是使用了 MultiInstanceComposite，它允许仅仅通过改变一个参数值来改变 Channel 实例的数量

如图 2-41 所示，MultiInstanceComposite 角色有 3 个参数：nInstances、instance、showClones。第一个参数指定了要创建的实例数量。运行时，该角色将自我复制，次数由 nInstances 指定，如第一个（原型）实例那样，将输入和输出连接到相同的源和目的地。在图 2-40 中，注意 MultiInstanceComposite 的输入连接到一个关系（黑色菱形），输出直接连接到一个 AddSubtract 角色的多端口输入。因此，连接多个实例的方式类似于图 2-32，其中相同的输入值将传递给所有实例，但是提供给 AddSubtract 角色的输出值是不同的。

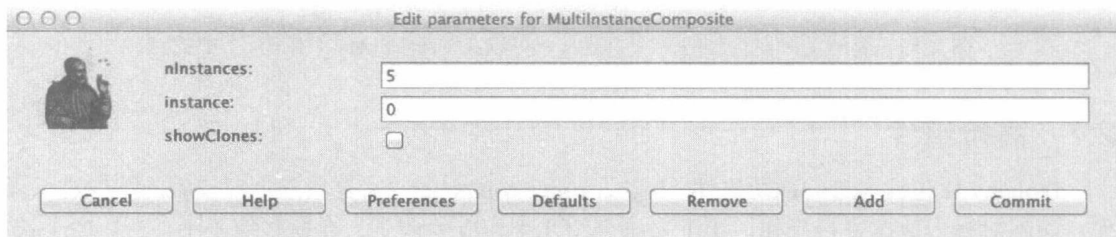


图 2-41 MultiInstanceComposite 的第一个参数指定实例的数量。第二个参数可以使模型的建立者识别单独的实例。第三个参数控制实例是否呈现在屏幕上（当模型运行时）

因为实例的数量可以由一个参数改变，所以使用多实例创建模型比原始方法更好。根据 MultiInstanceComposite 中的 instance 参数来表示每个实例中的参数值，每个实例都可以按需定制。比如，令图 2-40 中 InstanceOfChannel 角色的 noisePower 参数值取决于 instance。如将其设置为 $instance \times 0.1$ ，然后将 nInstance 设置为 1。当运行模型时，将看到清晰的正弦波。这是因为这个 instance 的值为 0，所以该实例中没有噪声。

2.7.2 IterateOverArray 角色

如图 2-37 所示，Channel 类的实现没有包含任何状态，这意味着信道模型的调用

不依赖于上次调用的数据计算。因此，没有必要使用 Channel 类的几个不同实例来实现一个多样化通信系统；一个实例可以对 n 个数据副本调用 n 次。这种方法可以通过使用 IterateOverArray 高阶角色来实现。

IterateOverArray 角色可以用类似于前面章节中的 MultiInstanceComposite 的方法来实现。也就是说，我们可以在其内填充一个 Channel 类实例，类似于图 2-40。IterateOverArray 角色在模型中也需要一个指示器。

例 2.2 考虑图 2-42 中的例子。在这种情况下，顶层模型使用一个含有信道输入的多副本数组，而不是使用一个关系将输入广播到 Channel 的多个实例。这是使用 Repeat 角色（在 FlowControl → SequenceControl 子库中找到）和 SequenceToArray 角色（见 2.7.3 节补充阅读）的结合来完成的。Repeat 有一个参数 numberOfTimes，在图 2-42 中将其值设置为等于 diversity 参数的值。SequenceToArray 角色有一个参数 arrayLength，其值也可以设置为等于 diversity 参数的值（这个参数可以通过 arrayLength 端口来设置，灰色填充表示它既是一个参数又是一个端口）。输出被发送到 ArrayAverage 角色（见 2.8 节补充阅读：处理数组的角色）。

图 2-42 中模型的执行与它前面的版本类似，除了输出的规模不同外，其输出是平均值而不是总和。

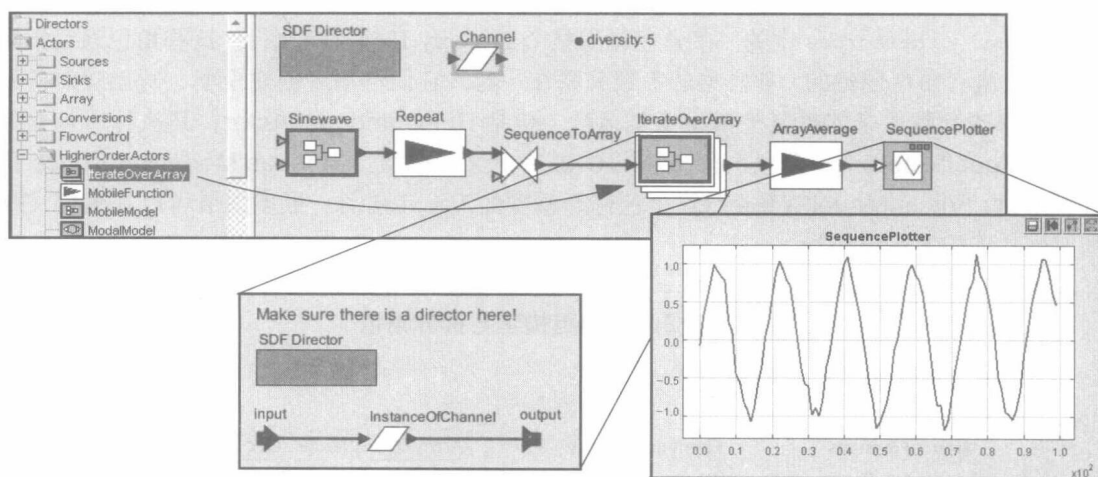


图 2-42 IterateOverArray 角色可以用来完成与图 2-40 中一样多的信道模型，但是不需要创造信道模型的多实例。这种方法是可能的，因为信道模型没有状态

不论 IterateOverArray 包含什么角色，对于输入数组的每个元素，都将进行简单重复的执行动作。如图 2-42 所示，它包含的角色可以是不透明的复合角色。然而，有趣的是，它也可以作为原子角色。为了使用带有 IterateOverArray 的原子角色，只要将原子角色拖到 IterateOverArray 实例中。然后，它将执行输入数组的每一个元素中的原子角色，并产生作为输出的结果数组。该机制将在图 2-43 中说明。当一个角色从库中拖出并移到 IterateOverArray 角色时，它的图标将获得白色的边框。这个边框表明，如果角色被放下，它将放在光标下的角色中，而不是模型所包含的角色上。放入的 IterateOverArray 角色将成为为输入数组的每一个元素执行的角色。为了和 Channel 角色一起使用，定义了如上规则，但是需要将 Channel 角色转换为一个不透明角色，方式是通过插入一个指示器，因为对于数组元素 IterateOverArray 只能应用不透明角色。

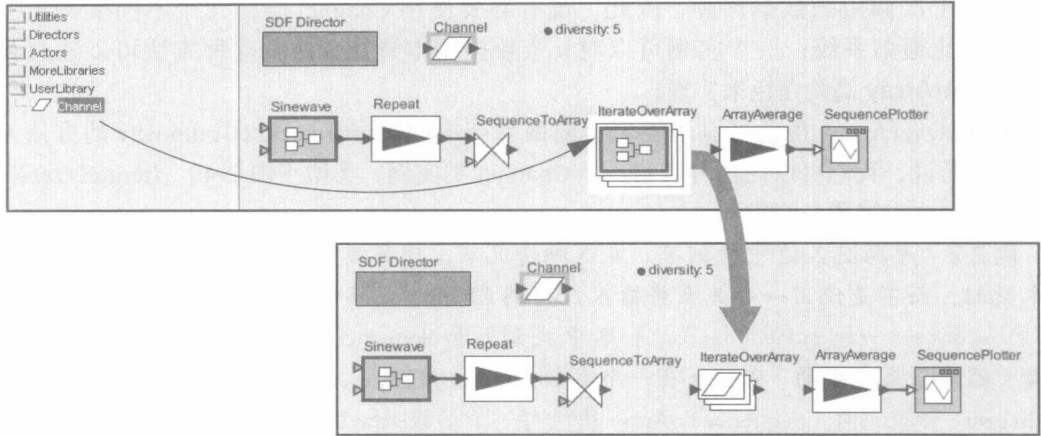


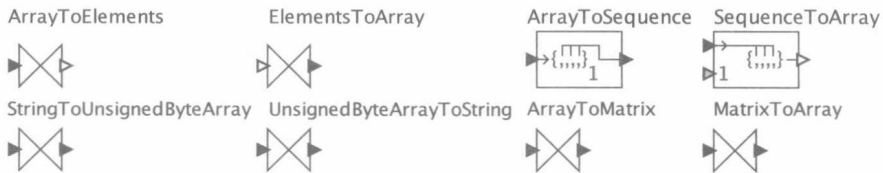
图 2-43 IterateOverArray 角色支持在它上面放入一个角色。它变换为模仿放入角色的图标。这里我们使用 Channel 类，保存到如图 2-39 所示的 UserLibrary (用户库) 中

2.7.3 生命周期管理角色

Higher OrderActors 中的一些角色调用整个 Ptolemy II 模型的执行。这些角色通常把端口 (用户或者角色创建的) 和模型的参数关联在一起。它们可用于创建模型，所创建的模型可通过修改参数值来使得其他模型反复运行。包括 RunCompositeActor，其执行所包含的模型。ModelReference 角色执行文件或 URL 中定义的模型。当 VisualModelReference 执行模型时，VisualModelReference 角色打开模型的 Vergil 视图。更多细节可以在角色文档和 Vergil 连接展示中找到。

补充阅读：数组构建与拆分角色

下面是构建和拆分数组的角色：



- ArrayToElements 将输出端口信道中的数组元素输出。
- ElementsToArray 用输入端口信道中的元素构建一个数组。
- ArrayToSequence 将输出端口中的数组元素顺序输出。
- SequenceToArray 用输入端口中的一系列元素构建一个数组。
- StringToUnsignedByteArray 从字符串构建一个数组。
- UnsignedByteArrayToString 从数组构建一个字符串。
- ArrayToMatrix 从数组中构建一个矩阵。
- MatrixToArray 从矩阵构建一个数组。

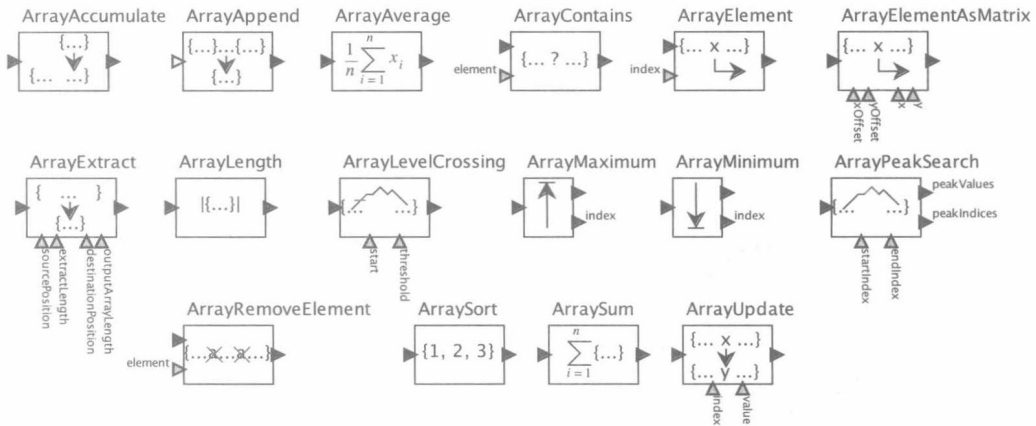
此外，很多多态角色，比如说 AddSubtract，也能对数组进行操作。

2.8 小结

本章介绍了名为 Vergil 的 Ptolemy II 可视化界面，它支持通过图形化方式构建模型，与此同时，还介绍了 Ptolemy II 系统的一些基础功能。后续章节将着重介绍各种可用的指示器的性质。附录部分将着重介绍计算模型的通用架构和跨计算模型的能力。

补充阅读：处理数组的角色

下面是角色可以对数组进行的操作：



- **ArrayAccumulate** 将输入数组附加于上一数组之后，以扩大输出数组。
 - **ArrayAppend** 添加多重端口的信道上提供的输入数组。
 - **ArrayAverage** 求数组元素的平均值。
 - **ArrayContains** 确定数组是否包含特定的元素。
 - **ArrayElement** 从一个数组中提取某个元素。
 - **ArrayElementAsMatrix** 使用类似矩阵的索引来提取元素。
 - **ArrayExtract** 提取子数组。
 - **ArrayLength** 将输入数组的长度输出。
 - **ArrayLevelCrossing** 找出超过阈值的元素。
 - **ArrayMaximum** 寻找数组中的最大元素。
 - **ArrayMinimum** 寻找数组中的最小元素。
 - **ArrayPeakSearch** 寻找数组元素的峰值。
 - **ArrayRemoveElement** 删除某个特定元素的实例。
 - **ArraySort** 对数组进行排序。
 - **ArraySum** 对数组元素求和。
 - **ArrayUpdate** 输出一个与输入数组类似的新数组，但替换原数组的一个元素。
- 另外，很多多态 (Polymorphic) 角色，如 **AddSubtract**，也可作用于数组。

补充阅读：移动代码

Ptolemy II 中的一些角色支持移动模型 (mobile model)。也就是说，对于从一个角色

传递到另一个角色的数据，与其说它是运行于模型上的数据，不如说它是个待执行的模型。**ApplyFunction** 角色从一个输入端口接收一个用表达式语言（见第 13 章）描述的函数，并将该函数作用于另一个端口（用户必须创建这个端口）到达的数据。**MobileModel** 角色从一个端口接收一个 Ptolemy II 模型的 MoML 描述，然后执行该模型来处理从另一端输入的数据流。

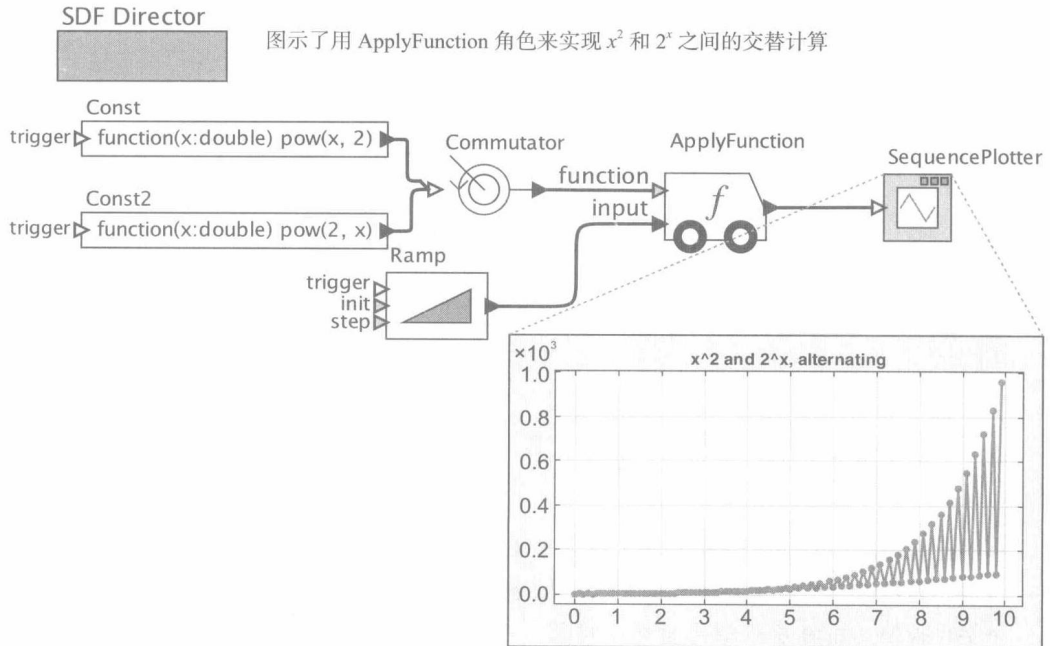


图 2-44 ApplyFunction 角色从一个端口接收函数定义，并将函数作用于到达另一个端口的数据

ApplyFunction 角色的使用见图 2-44。在该模型中，以交替方式为 ApplyFunction 角色提供两个函数：一个函数计算 x^2 ，另一个函数计算 2^x 的。这两个函数由两个 Const 角色提供，可以从 Sources → GenericSources 子库中找到。这些函数由 Commutator 角色交替输出，Commutator 在 FlowControl → Aggregators 子库中。

计算模型

本文第二部分主要介绍一些常用于系统设计、建模和仿真的计算模型。虽然不是一个很全面的集合，但也是一个比较有代表性的集合。这里提到的计算模型（Model of Computation, MoC）是相对成熟、容易理解并完全可以实现的计算模型。这些计算模型在 Ptolemy II 之外的工具上大部分也是可以实现的，但除 Ptolemy II 外没有工具可全部实现它们。

数 据 流

Edward A. Lee、Stephen Neuendorffer 和 Gang Zhou

Ptolemy II 能够使异构系统的开发和仿真一同进行，将开发和仿真作为整个系统建模的一部分。正如前两章讨论的那样，不同于其他设计和建模环境，Ptolemy II 的一个关键创新在于支持多种计算模型，这些计算模型可被剪裁以适应具体的建模问题。这些计算模型定义模型的行为，计算模型的种类由模型中使用的指示器决定。在 Ptolemy II 的术语中，指示器实现了一个域，这是对计算模型的实现。因此，监视器、域和计算模型就被绑定在一起。比如说，当构建一个包含 SDF 指示器（一种同步数据流指示器）的模型时，实际上已经通过 SDF 计算模型建立了一个“SDF 域”模型。

本章对当前 Ptolemy II 中的可用数据流域（dataflow domain）进行描述，其包含同步（静态）数据流模型和动态数据流模型。数据流域适用于那些涉及数据值流处理的应用，这些数据值流在一系列角色间流动，角色以一定的方式进行数据值流的转化。这些模型通常叫作管道和滤波器（pipe and filter）模型，因为角色之间的连接类似于承载流的管道，并且角色就像对这些流做了一些改变的滤波器。数据流域几乎是忽略时间的，尽管同步数据流（SDF）有能力在迭代之间建立有均匀时间间隔的流模型。后续的章节将讨论其他的域，以及其选择和使用。

3.1 同步数据流

同步数据流（Synchronous DataFlow, SDF）域，也称为静态数据流（static dataflow）^①，由 Lee and Messerschmitt (1987b) 提出，是 Ptolemy II 最先开发的域（或者计算模型）之一。它是数据流模型的一种特殊类型。在数据流模型中，当角色需要的数据输入并可用时，角色开始执行（其被点火）。SDF 是一种相对简单的数据流，角色的执行顺序是静态的，不依赖被处理的数据（令牌的值在角色之间传递）。

在一个同构 SDF（homogeneous SDF）模型中，当角色的每一个输入端口都有令牌时，角色被点火，并在每一个输出端口产生一个令牌。在这种情况下，指示器仅需确保当角色获得数据后再点火，并且模型中的一次迭代包含每个角色的一次点火。第 2 章的大多数例子就是同构 SDF 模型。

然而不是所有的角色每次点火都只会产生和消耗一个令牌，某些角色在它们点火前需要多个输入令牌并产生多个输出令牌。负责决定角色执行顺序的 SDF 调度程序支持比同构

① “同步数据流”这个词会引起混淆，因为它并不是第 5 章中 SR 的那种意义的同步。在 SDF 模型中没有全局时钟，并且角色是异步点火的。出于这个原因，有些作者更喜欢“静态数据流”这个词。然而，这并没有避免所有的混淆，因为 Dennis (1974) 曾经创造了“静态数据流”这个词来指代数据流图，该数据流图中的缓冲区最多持有一个令牌。既然没有办法避免术语的冲突，因此在文献中坚持原来的“同步数据流”术语使用。SDF 一词源于信号处理的概念，即采样率呈有理倍数关系的两个信号被认为是同步的。

SDF 更复杂的模型。只要指定每个端口上的每个角色点火所产生和消耗的令牌数量来给出这些速率，SDF 调度器就能够对任意数据速率角色的执行进行调度。

例 3.1 有些角色需要多个输入令牌才能点火，Spectrum 角色就是这样一个例子。如图 3-1 所示为一个可以计算带噪声正弦波的频谱的系统，该系统与图 2-20 中构造的相同。Spectrum 角色有一个单独的参数，用来指定快速傅里叶变换 (FFT) 的采样点位数 (order)。快速傅里叶变换用来进行频谱的计算。如图 3-1 显示了 order 设置为 8 以及迭代次数设置为 1 的模型的输出 (见 17.2 节)。

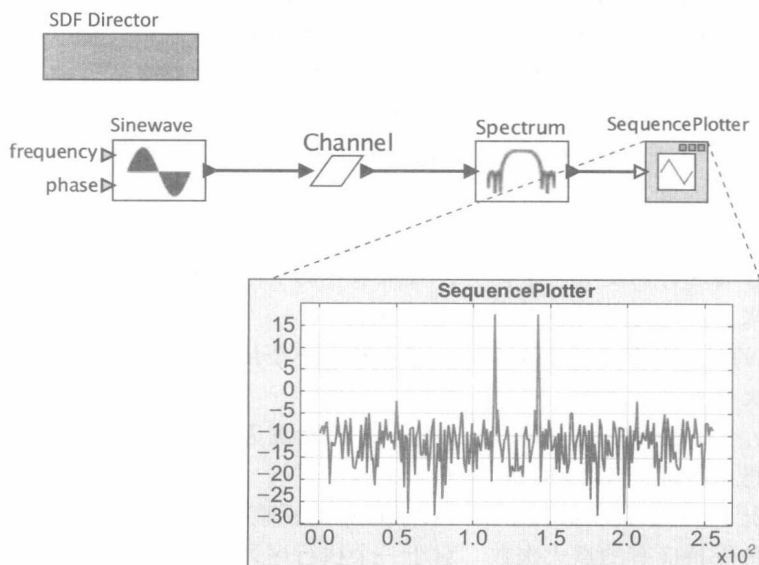


图 3-1 一个多速率 SDF 模型。Spectrum 角色需要 256 个令牌才能点火，所以这个模型的迭代需要 256 个 Sinewave、Channel、SequencePlotter firings 角色的点火以及一次 Spectrum 角色的点火

当 order 参数设置为 8 时，Spectrum 角色的点火需要 2^8 (256) 个输入样本，并产生 2^8 个输出样本。为了使 Spectrum 角色点火一次，为它提供输入数据的 Sinewave 和 Channel 角色必须各自点火 256 次。SDF 指示器提取该关系，并定义模型的一个迭代由 Sinewave、Channel、SequencePlotter 角色的 256 个点火行为和 Spectrum 角色的一个点火行为组成。

这个例子实现了一个多速率 (multirate) 模型，即角色的点火速率并不是完全相同的。特别是，Spectrum 角色的执行速率比其他角色慢。对于多速率模型的执行，有且只有一次迭代是很常见的。在下一节，指示器将使用平衡方程来决定每次迭代中各个角色的点火次数。

3.1.1 平衡方程

考虑 A 和 B 两个角色之间的单一连接，如图 3-2 所示。图中符号表示当 A 点火时，它在输出端产生 M 个令牌，当 B 点火时，它在输入端消耗 N 个令牌。 M 和 N 是非负整数。 A 点火 q_A 次， B 点火 q_B 次。

当且仅当满足下面的平衡方程时， A 所产生的令牌才能都被 B 消耗掉，



图 3-2 SDF 角色 A 点火时产生 M 个令牌，角色 B 点火时消耗 N 个令牌

$$q_A M = q_B N \quad (3-1)$$

给定的 q_A 和 q_B 满足式 (3-1) 时, 系统保持平衡, 即 A 产生的令牌与 B 消耗的令牌数量一样。

假设希望能处理任意数量的令牌, 典型的流应用就是此类情况。一个简单的策略是点火角色 Aq_A 次 (q_A 为任意大的数) 并点火角色 Bq_B 次, 其中 q_A 和 q_B 满足式 (3-1)。然而, 该策略是一个低级策略, 因为它需要在一个缓冲区中存储一个任意大数量的未消耗令牌。一个较优的策略是求出满足式 (3-1) 的最小正整数 q_A 和 q_B 。然后建立一个调度表, 点火角色 Aq_A 次, 角色 Bq_B 次, 并且也可以多次重复调度, 而不需要更大的存储空间来存储未消耗的令牌。也就是说, 可以在有界缓冲区 (bounded buffer) (缓冲区中未消耗的令牌数量有限) 内完成无限的执行 (unbounded execution) (处理任意数量令牌的执行)。在每轮调度 (被称为一次迭代) 中, 角色 B 消耗的令牌数量与角色 A 产生的令牌数量恰好一致。

例 3.2 假设在图 3-2 中, $M=2$, $N=3$ 。这里有多种可能解来匹配平衡方程, 其中一种是 $q_A=3$ 和 $q_B=2$ 。根据这些值, 可以重复以下调度:

A, A, A, B, B

另一个调度也可以使用:

A, A, B, A, B

事实上, 后面这种调度有一个优势, 它用来存储中间令牌的存储器更少。一旦有足够的令牌 B 就会点火, 而不是等待 A 完成整个循环。

式 (3-1) 的另一组解是 $q_A=6$ 和 $q_B=4$ 。与上一个使系统保持严格平衡的解相比, 这组解包含了更多次的点火行为。

$q_A=0$ 和 $q_B=0$ 也满足式 (3-1), 但是如果角色点火的数量是零, 就会无法完成工作。显然, 这不是想要的解, 负的解也是毫无意义的。

在默认情况下, SDF 指示器为平衡方程求最小的正整数解, 并且依据该组解构建调度表, 使模型中的角色保持应有的点火次数。对于一个执行序列, 若每个角色的点火次数都与方程解出的结果完全一致, 则称为一次完整迭代 (complete iteration)。

在更复杂的 SDF 模型中, 每个角色之间的连接都会导致一个平衡方程。因此, 模型定义一个方程组并求最小正整数解是重要的。

例 3.3 图 3-3 展示了一个由 3 个 SDF 角色组成的网络。这些连接产生了如下平衡方程组:

$$q_A = q_B$$

$$2q_B = q_C$$

$$2q_A = q_C$$

这些方程的最小正整数解是: $q_A=q_B=1$, $q_C=2$, 所以下面的调度可以一直重复从而得到一个有界缓冲区内无限执行的情况:

A, B, C, C

平衡方程并不是总有非零解, 如下所述。

例 3.4 图 3-4 显示了一个有 3 个 SDF 角色的网络, 其平衡方程的唯一解是一个无效解, $q_A=q_B=q_C=0$ 。结果就是, 对于该模型来说, 没有可以在有界缓冲区内进行无限执行的可能。即它不能保持平衡。

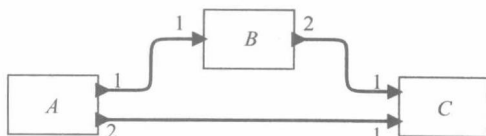


图 3-3 一个一致的 SDF 模型

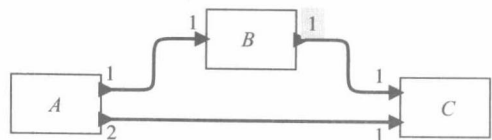


图 3-4 一个非一致的 SDF 模型

拥有平衡方程非零解的 SDF 模型称为一致的 (consistent)。如果唯一解是零解，那么它是不一致的 (inconsistent)。一个不一致的模型不存在在有界缓冲区内进行无限执行的情况。

Lee and Messerschmitt (1987b) 说明如果平衡方程有非零解，那么它也总有一个对于所有的角色 i , q_i 都是非负整数的解。此外，对于连接模型 (其中任意两个角色之间都有通信路径)，他们给出了求最小正整数解的过程。该过程形成了 SDF 模型调度器的基础。

例 3.5 图 3-5 显示了一个 SDF 模型，它大量使用了 SDF 的多速率功能。模型在机器上运行，AudioCapture 角色从机器的麦克风捕获声音，在每秒 8000 次采样的默认速率下产生一系列样本。Chop 角色从包含 500 个样本的输入块中提取 128 个样本块 (见 3.1.3 节补充阅读)。Spectrum 角色计算功率谱，得到功率随着频率变化的情况，即功率是频率的函数。两个 SequenceToArray 角色构造数组，然后 ArrayPlotter 角色将数组用图形表示 (见第 17 章)。图中两个特殊点是对哨声的响应。注意，频谱的峰值约出现在 1700Hz 和 -1700Hz 处。

模型中的 SDF 指示器计算出每点火一次 Chop、Spectrum、SequenceToArray 和绘图仪都需要点火 AudioCapture 角色 500 次。

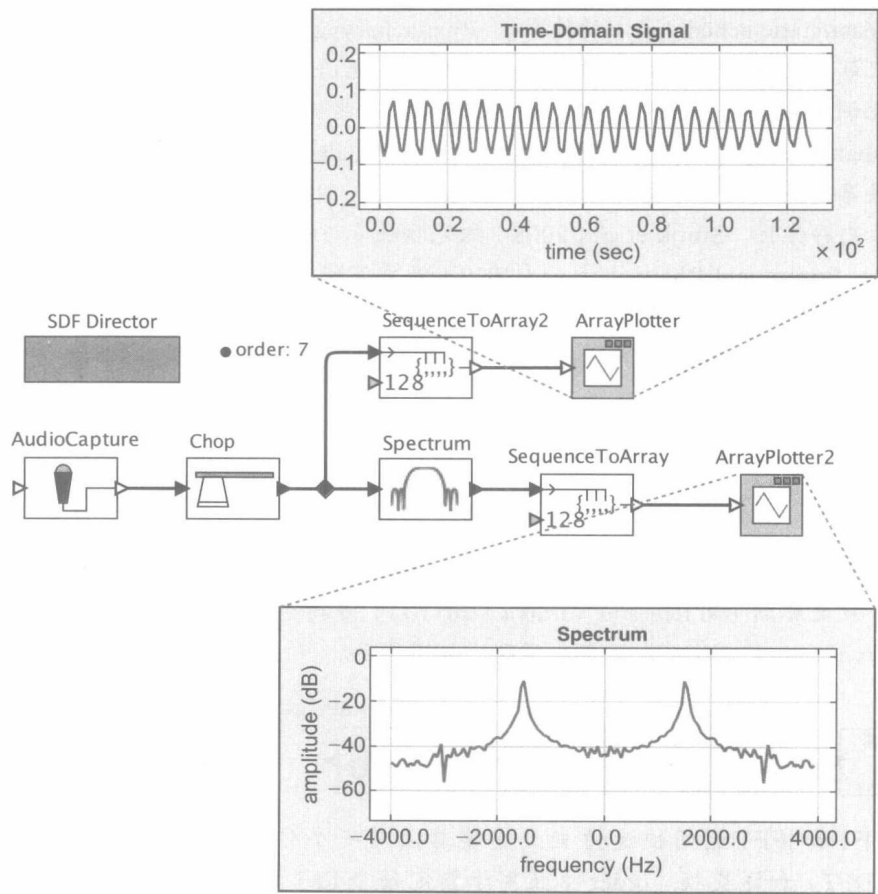


图 3-5 该模型从麦克风获取声音信号并计算其功率谱。此图展示了一个频率大约为 1700Hz 的哨声功率谱

补充阅读：SDF 调度器

使用 SDF 的一个关键优势是，对于包含一些并行执行角色的给定模型，有多种可能的调度。在这种情况下，数据流图中的角色可以被映射到多核结构或者分布式体系结构中不同的处理器中以便提高性能。Lee and Messerschmitt (1987a) 通过把 SDF 图转换为一个无环优先图 (APG)，将一些经典作业车间调度算法 (Coffman, 1976)，特别是 Hu (1961) 提出的那些算法，运用在 SDF 中。Lee and Ha (1989) 将调度策略分为完全动态调度 (fully dynamic scheduling) (所有调度决策在运行时完成)、静态分配调度 (static assignment scheduling) (除了处理器分配的决策外其他所有决策都在运行时完成)、自定时调度 (self-timed scheduling) (只有角色点火的时间安排在运行时决定) 和完全静态调度 (fully-static scheduling) (调度的每个方面都在运行时前完成)。Sih and Lee (1993a) 对作业车间调度技术进行了解释，用以说明解释处理器之间的通信开销 (见 Sih and Lee (1993b))。Pino et al. (1994) 展示了如何构建异构多处理器的调度。Falk et al. (2008) 给出了一个基于聚类的并行调度策略，并说明了该策略在多媒体应用中可带来显著的性能提升。

除了并行调度策略外，其他的调度优化也很有用 (见 Bhattacharyya et al. (1996b))。Ha and Lee (1991) 放松了 SDF 的限制，允许数据依赖的角色迭代点火 (一种称为准静态调度 (quasi-static scheduling) 的技术)。Bhattacharyya and Lee (1993) 为角色迭代调用提出优化调度策略 (也见 Bhattacharyya et al. (1996a))。Bhattacharyya et al. (1993) 提出的优化调度主要减少对内存的使用，随后并将这些优化应用到嵌入式处理器的代码生成中 (Bhattacharyya et al., 1995)。Murthy and Bhattacharyya (2006) 收集通过调度和缓冲区共享来最小化内存使用的算法。Geilen et al. (2005) 的研究表明，模型检测技术可以用于内存优化。Stuijk et al. (2008) 探究吞吐量与缓冲之间的折中 (见 Moreira et al. (2010))。Sriram and Bhattacharyya (2009) 开发了减少并行 SDF 中同步操作数量的调优化度。同步确保一个角色在它接收需要点火的数据前不会被点火，然而，如果一个之前的同步已经确保数据准时到达，那么该同步就不需要了。通过操作调度，可以减少所需同步点的数量。

补充阅读：频率分析

SDF 域对信号处理特别有效。信号分析的一种基本操作方式是将时域信号转换成频域信号，反之亦然 (见 Lee and Varaiya (2011))。支持这一操作的角色可以在 Actors → SignalProcessing → Spectrum 库中找到，如下所示。



- FFT 和 IFFT 利用快速傅里叶变换算法对一个信号分别进行离散傅里叶变换 (DFT) 和逆变换。order 参数用于指定每个 FFT 计算的输入令牌数。它是“基 2”算法，这意味着需要的令牌数是 2 的幂，且 order 参数指定指数的值。比如，如果 order=10，那么用于每个点火的输入令牌数就是 $2^{10}=1024$ 。剩下的角色可

实现多种频谱估计算法，它们都是使用 FFT 作为组件的复合角色。这些算法以分贝 (dB) 为单位输出信号功率，作为频率的函数。输出频率范围为 $-f_N \sim f_N$ ，其中 f_N 是奈奎斯特频率 (采样频率的一半)。也就是说，最先输出的一半代表负频率，余下的一半代表正频率。

- **Spectrum** 是最简单的谱估计器。它计算输入信号的 FFT，并将结果转换为以分贝计量的功率。
- **SmoothedPeriodogram** 通过首先估计输入的自相关性来计算功率谱。该方法对输入进行平均，并且对噪声不敏感。
- **MaximumEntropySpectrum** 是一个参数谱估计器。它使用 Levinson-Durbin 算法构造回归 (AR) 模型的参数，该模型能合理地产生输入信号。然后选择最大化熵的模式 (见 Kay (1988))。它是最复杂的谱估计器，通常产生最平滑的估计。

如图 3-6 所示为 3 个谱估计器输出的比较，其中输入由 3 个含噪声的正弦曲线组成。为一个应用选择正确的谱估计器是个复杂的主题，超出了本书的范围。

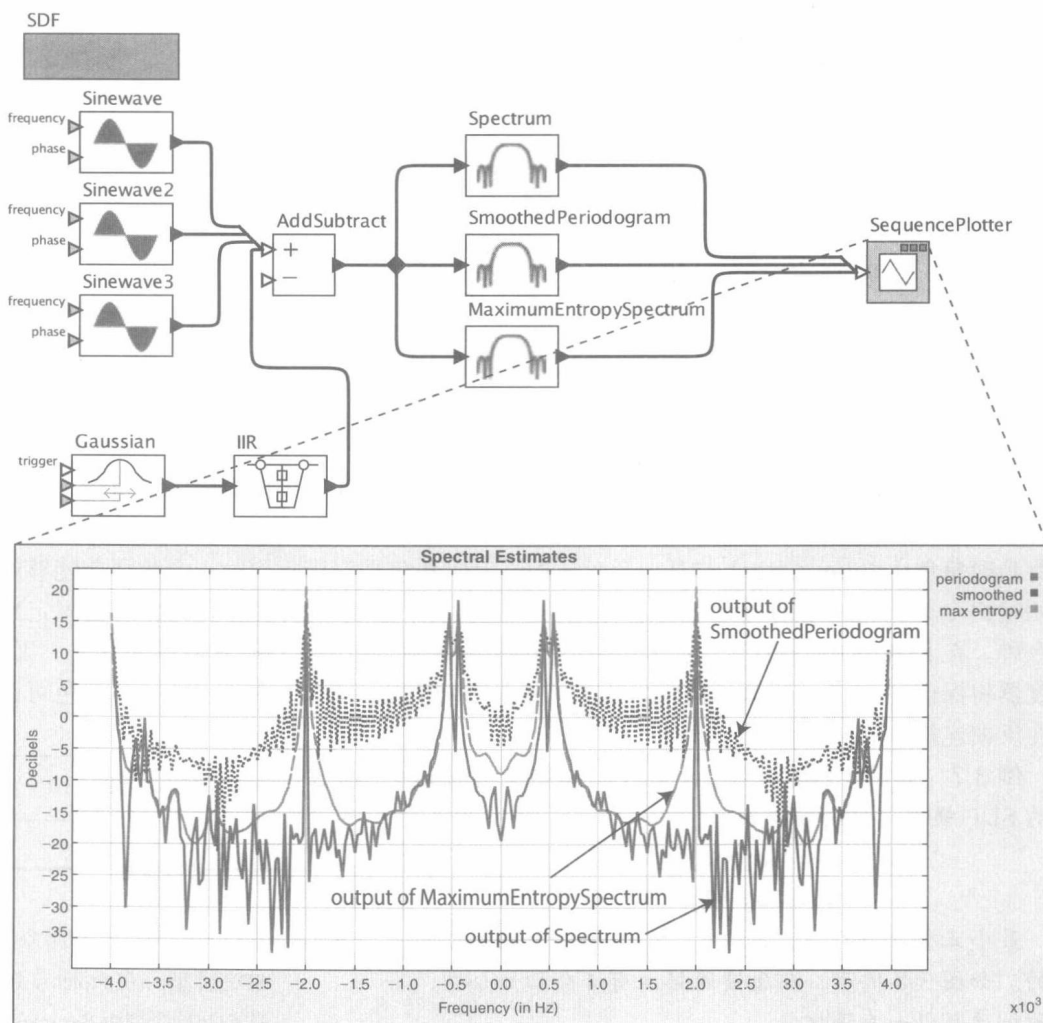


图 3-6 Spectrum、SmoothedPeriodogram 和 MaximumEntropySpectrum 谱估计技术的比较

3.1.2 反馈回路

在 SDF 中，一个反馈回路必须至少包含一个 **SampleDelay** 角色的实例（在 **FlowControl** → **SequenceControl** 子库中）。若没有该角色，回路将产生死锁（deadlock），即反馈回路中的角色将无法点火，因为它们依赖彼此的令牌。**SampleDelay** 角色通过在模型开始点火前在它的输出端产生初始令牌来解决这一问题。初始令牌通过 *initialOutputs* 参数来指定，该参数定义了一个令牌数组。这些初始令牌能使下游角色点火，并打破无初始令牌时将在反馈回路中的循环依赖。

例 3.6 如图 3-7 中所示的模型。为一个同构 SDF 模型，其使用反馈回路产生一个计数序列。**SampleDelay** 角色通过在它的输出能产生一个值为 0 的令牌来开始该过程。这个令牌与从 **Const** 角色来的一个令牌一起，能使 **AddSubtract** 角色点火。该角色的输出能使下一个 **SampleDelay** 点火。在初始点火之后，**SampleDelay** 角色将输入无修改地复制到输出。输入到它的输出不变。

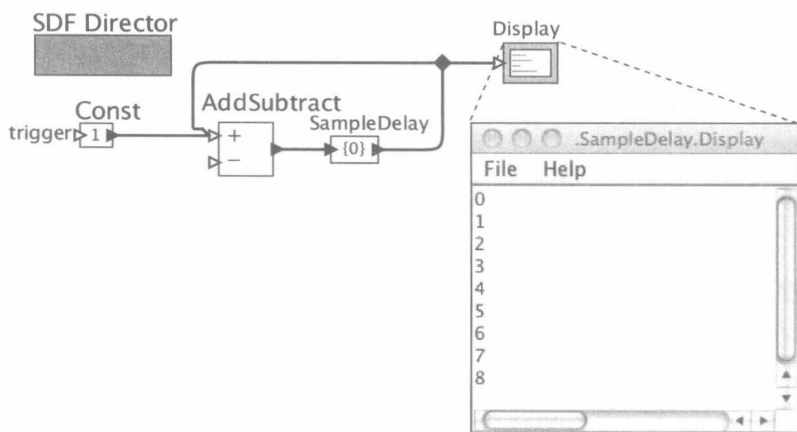


图 3-7 带反馈回路的 SDF 模型其中必须至少有一个 **SampleDelay** 角色的实例

一致性对保证缓存区有界是充分条件，但是并不足以保证一个模型的无限执行。即使模型是一致的，它也可能产生死锁。SDF 指示器分析模型的一致性和死锁。为了允许反馈，它对待**延时角色（delay actor）**与其他角色不同。延时角色在它接收到输入令牌之前能够产生初始输出令牌。它之后的行为与普通的 SDF 角色类似，在每次点火时消耗与产生固定数量的令牌。在 SDF 域中，初始令牌理解为执行的初始条件，而不是执行本身的一部分。因此，调度器将保证在 SDF 执行开始前产生所有的初始令牌。从概念上，**SampleDelay** 角色可以被放在反馈连接上的初始令牌代替。

例 3.7 如图 3-8 所示为一个在反馈回路上有初始令牌的 SDF 模型，平衡方程为：

$$3q_A = 2q_B$$

$$2q_B = 2q_A$$

最小正整数解存在，是： $q_A=2$ 、 $q_B=3$ ，因此模型是一致的。如图 3-8 所示，在反馈连接上有 4 个初始令牌，下面的调度可以一直迭代

$$A, B, A, B, B$$

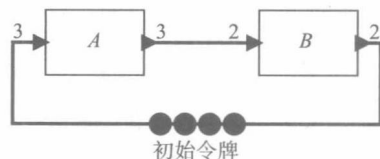


图 3-8 在反馈回路上有初始令牌的 SDF 模型。在 Ptolemy II 中，这些初始令牌由 **SampleDelay** 角色提供

这个调度从角色 *A* 开始，因为在执行的开始；仅有角色 *A* 可以点火，因为角色 *B* 没有足够的令牌。当 *A* 点火时，它从 4 个初始令牌中消耗 3 个令牌，留下 1 个令牌。它传送 3 个令牌给 *B*。此时，仅有 *B* 可以点火，消耗由 *A* 发送来的 3 个令牌中的 2 个。并且又提供 2 个令牌给它的输出。此时，角色 *A* 可以再次点火，因为它的输入端有 3 个令牌。它将消耗所有令牌并产生 3 个令牌。在此，*B* 在它的输入拥有 4 个令牌，能够点火 2 次。在那 2 次点火之后，2 个角色都已经点火了必要的次数，并且反馈缓冲区再一次拥有了 4 个令牌。调度由此将数据流图返回到它的初始情况。

然而，若在反馈路径上的初始令牌少于 4 个，模型就会死锁。例如，如果只有 3 个令牌，那么 *A* 可以在 *B* 之后点火，但是不会有足够的输入令牌使其再次点火。

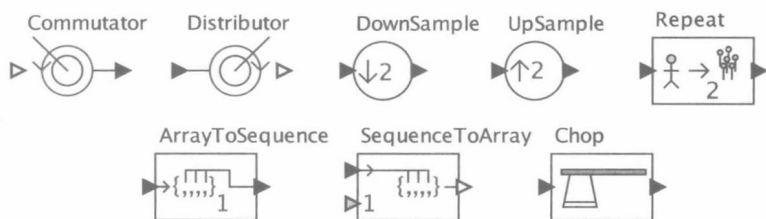
Lee and Messerschmitt (1987b) 讨论了求解平衡方程的过程，还讨论了一个要么为无限执行提供调度，要么证明这样的调度不存在的过程。通过这些过程，SDF 模型的有界缓冲区和死锁就可解决（这意味着，Ptolemy 可以确定在任何 SDF 模型中是否会发生死锁或者存在无界缓冲区）。

3.1.3 数据流模型中的时间

到目前为止，在已示的 SDF 例子中，使用了 SequencePlotter 角色，而不是 TimedPlotter 角色（见第 17 章）。这是因为 SDF 域通常不使用其模型中的时间概念。在默认情况下，时间不会随着 SDF 模型的执行而推进（即使 SDF 指示器确实包含了一个称为 period 的参数，该参数可以通过在模型的每次迭代中的一个固定量来推进时间）。因此，在大多数的 SDF 模型中，TimedPlotter 角色显示的时间轴通常等于零。相比之下，SequencePlotter 角色进行一个非基于时间值序列的描述，因此经常用于 SDF 模型。在第 7 章和第 9 章中讨论的离散事件和连续域包含了大量的更强的时间概念，因此经常使用 TimePlotter。

补充阅读：多速率数据流角色

Ptolemy II 的库提供了一些产生“与 / 或”的角色，它们在每点火一次需要消耗多个令牌。最基本的一些如下所示：



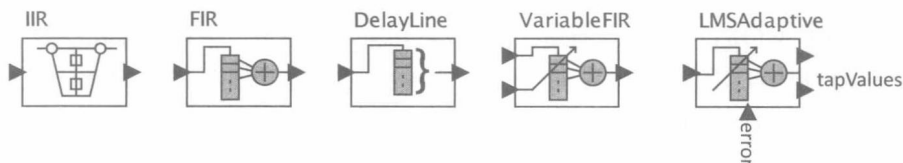
- **Commutator** 和 **Distributor** 在 FlowControl → Aggregators 子库中，将来自多个信号的令牌转换为一个令牌序列，反之亦然。Commutator 有一个多重端口输入，在每次点火时，它从每个输入通道中读取固定数量的令牌（由其 blockSize 参数给出），并把所有来自输入通道的令牌输出为一个序列。Distributor 则为其逆过程。
- **DownSample** 和 **UpSample** 位于 SignalProcessing → Filtering 中，丢弃或者插入令牌。Downsample 读取固定数量的令牌（由它的 factor 参数给出），并输出其中一个令牌（由 phase 参数选择）。UpSample 在输入令牌之间插入固定数量的零

值令牌。

- **Repeat** 位于 `FlowControl` → `SequenceControl` 中，除了用迭代方式输入令牌代替插入零值令牌外，其他方面类似于 `UpSample`。
- **ArrayToSequence** 和 **SequenceToArray** 位于 `Array` 库中，将数组令牌转换为一个令牌序列，反之亦然。两个角色都有一个 `arrayLength` 参数，它指定传入（或传出）数组的长度。`ArrayToSequence` 也有一个 `enforceArrayLength` 参数，如果其设置为真，将导致在接收到一个错误长度的数组时角色将产生一个错误信息。在 `SequenceToArray` 中，`arrayLength` 是端口参数（`PortParameter`），因此读取的输入令牌数量可以不同。仅当数组长度为常数时，这些角色是 SDF 角色。
- **Chop**，在 `FlowControl` → `SequenceControl` 中，读取指定数量的输入令牌并产生这些输入的特定子集，可以用零值令牌或者以前消耗的令牌填充。

补充阅读：信号处理角色

除了前文描述的频谱分析角色，Ptolemy II 还包含了其他几个信号处理角色，如下所示。



- **IIR** 实现无限脉冲响应滤波器，也称为递归滤波器（见 Lee and Varaiya (2011)）。滤波器系数由两个数组提供，一个提供传输函数的分子多项式，一个提供传输函数的分母多项式。
- **FIR** 实现有限脉冲响应滤波器，也称为抽头延迟线滤波器，它的系数由 `taps` 参数来指定。而 **IIR** 是同构 SDF（单速率）角色，**FIR** 是内在的多速率角色。当 `decimation`（`interpolation`）参数不等于 1 时，滤波器的行为就好像在使用 `DownSample`（`UpSample`）角色之后（之前）。然而，该实现比使用 `UpSample` 或 `DownSample` 角色实现会更高效，内部使用了多相结构，避免了不必要的内存使用和乘零操作。用这种方法通过合理的因素可以完成对任意样本速率的转换。
- **DelayLine** 产生一个数组而不是像 **FIR** 那样产生一个标量。**DelayLine** 不输出加权平均延迟线的内容（这是 **FIR** 产生的），而是仅仅简单地输出延迟线为一个数组。
- **VariableFIR** 与 **FIR** 相同，除了系数是由输入端口（因此可以改变）的数组提供而不是由角色参数来定义外。
- **LMSAdaptive** 类似于 **FIR**，除了在每次点火时使用梯度下降自适应滤波算法来调整系数外，该算法试图在错误的（`error`）输入端口将信号的功率最小化。

除了这里描述的角色外，信号处理（`signal processing`）库还包括固定点（`fixed`）和自适应格型（`lattice`）滤波器、统计分析角色等、通信系统的角色（如源和通道编码器与解码器）音频采集和回放、图像和视频处理角色等。

补充阅读：动态变化速率

SDF 的变体，称为参数化的 SDF (PSDF)，由 Bhattacharya and Bhattacharyya (2000) 提出，它端口的产生速率和消耗速率是由参数给出，而不是常数。参数的值允许改变，但仅仅是在两次完整的迭代之间改变。当这个参数的值改变时，新的调度必须用于下一次完整的迭代。

图 3-9 中的例子说明如何用 Ptolemy II 中的 SDF 指示器实现 PSDF。首先，注意指示器的 allowRateChanges 参数已经设置为真。这表明指示器可能需要计算不止一个调度，因此在执行模型期间 (rate) 速率参数可能改变。

其次，注意 Repeat 角色的 numberOfTimes 参数设置为与模型参数 rate 相等，其初始值为零。因此，当模型执行它的第一次迭代时，Repeat 角色将产生零个令牌，因此 Display 角色不会点火。Ramp 角色的初始输出为 1，将不会被显示。

在第一次迭代期间，Expression 和 SetVariable 角色都只点火一次。Expression 角色设置它的输出等于输入，除非它的输入等于 iterations (迭代) 参数 (它没在这个首次迭代中) 的值。SetVariable 角色设置 rate 参数的值为 1。在默认情况下，SetVariable 有一个 delayed 参数的值为真，这意味着只有在当前迭代完成之后 rate 参数才能改变。

在第二次迭代中，rate 参数的值是 1，因此 Repeat 角色复制一次它的输入 (值为 2) 给输出。Expression 和 SetVariable 角色把现在的 rate 参数设置为 2，因此在第三次迭代中，Repeat 角色复制它的输入 (有值为 3) 两次给它的输出。因此显示的输出序列为 2, 3, 3, 4, 4, 4...

为了终止模型，指示器的 iterations 参数设置为 5。在最后一次执行迭代中，Expression 角色保证 rate 参数重置为 0。因此，在下一次模型执行时，它将会随着 rate 参数设置为 0 重新开始。

在这个例子中，每次 rate 参数变化，SDF 指示器都会重新计算调度。在 PSDF 更好的实现中，它在计算调度前预计算调度与缓存，但是这个实现做不到预计算与缓存。它仅仅在迭代之间重新计算调度。

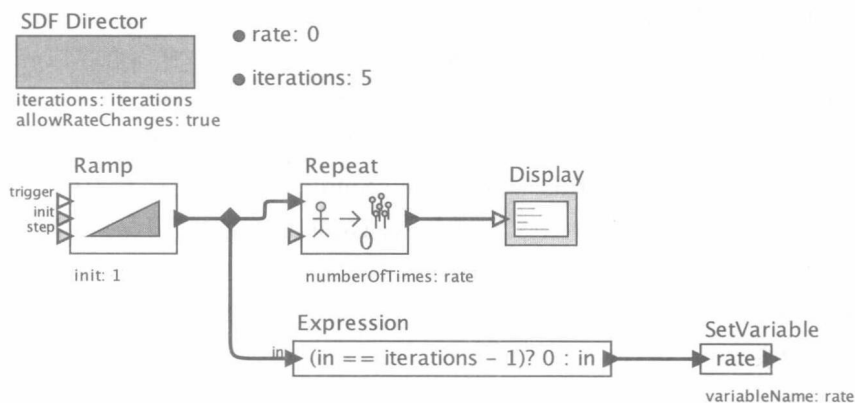


图 3-9 一个有动态变化速率的 SDF 模型

补充阅读：StreamIt

Thies et al. (2002) 给出一种基于 SDF 的文本程序语言——StreamIt，该语言主要针

对使用流数据的应用（如多媒体）。软件组件（称为**滤波器**而不是角色）产生和消耗固定数量数据。该语言为组成角色的常见模式提供紧凑的结构设计，如滤波器链、并行滤波器链和反馈回路等。

StreamIt 中的一个关键创新是**消息传递**（teleport message）的概念（Thies et al., 2005）。消息传递通过允许一个角色偶尔地发送一个消息给另一个角色来提高 SDF 的表达力。也就是说，不是每一个点火行为都发送消息，而是某些点火行为发送消息。尽管消息传递机制采用以下方式：当发送角色每次点火发出消息时，接收角色会产生同样的点火行为，确保该信息被接收。但是它避免了为每个点火行为发送消息的开销。这种方法建立了一个通信通道模型，其中令牌的产生和消耗不是总存在的而是偶尔存在的。但是它保留了 SDF 模型的不确定性，任何有效调度执行的结果都相同。

补充阅读：其他数据流的变体

SDF 的一个缺点是每个角色必须产生和消耗固定数量的数据。生产和消耗的速率不能取决于数据。DDF（3.2 节）取消了这一限制，代价是不能再对点火进行静态调度。此外，正如前面章节中所讨论的，分析所有模型的死锁和有界缓冲区已经不再可能了（这些问题是不可判定的。但是，研究人员已经开发了大量的数据流变种，它们比 SDF 更富有表达力，但仍然服从某些静态分析的形式。

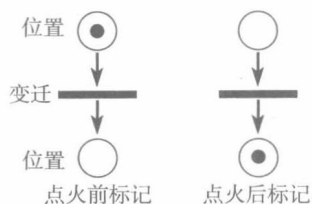
循环静态数据流（Cyclo-static DataFlow, CSDF）允许产生和消耗的速率呈周期性变化（Bilsen et al., 1996）。一个例子是 **SingleTokenCommutator** 角色（在 FlowControl → Aggregators 中）。这个角色类似于 Commutator 角色，但是它不是在单次点火行为中消耗所有输入，而是一次点火仅消耗其中一个通道的输入，并且通过在连续点火行为中的输入通道旋转。对于每个输入通道，消耗速率在 0 和 1 之间交替。这个角色在反馈系统中很有用，其中第二通道的输入依赖于第一通道的输入。

SDF 也可以分层地结合有限状态机（FSM）来创建模态模型，这将在第 8 章介绍。有限状态机的每个状态与子模型相关联（其中每个模型的细化都可以有不同的产生和消耗速率）。如果有限状态机的状态转换被限制为只发生在某些时刻，那么该模型仍然是确定的。这种结合由 Girault et al.（1999）提出，他称之为**异步数据流**（Heterochronous DataFlow, HDF）。SDF Scenarios（Geilen and Stuijk, 2010）类似于 HDF，因为它们也使用一个 FSM，但不是模型细化，在 SDF Scenarios 中有限状态机的每个状态与一组单一 SDF 模型的产生和消耗相关联。Bhattacharya and Bhattacharyya（2000）提出了参数化的 SDF（PSDF），其中产生和消耗速率可以依赖输入数据，同样的依赖关系可以在参数化的调度中表示。

Murthy and Lee（2002）介绍了**多维 SDF**（Multidimensional SDF, MDSDF）。MDSDF 中的一个通道传递一个令牌的多维数组，而 SDF 中的一个通道传递一个令牌序列。也就是说，令牌的记录可以增加为多个维度。这个模型对表示确定种类的信号处理应用是很有效的，特别是图像处理、音频处理、雷达和声呐。

补充阅读：Petri 网络

Petri 网，以 Carl Adam Petri 的名字命名，是一种流行的与数据流相关的建模形式 (Murata, 1989)。它们有两种类型的元素，库所 (place) 和变迁 (transition)，分别用白色圆圈和矩形表示。一个库所可以包含任意数量的令牌，表示为黑色圆点。如果所有的库所都连接到变迁，则变迁将被使能，它的输入至少包含了一个令牌。



一旦变迁被使能，它可以被点火 (fire)，从每个输入的库所消耗一个令牌，并在每个输出库所存放一个令牌。网络的状态，称为标记 (marking)，是网络中的每个库所上令牌的数量。上图显示了一个在点火行为之前和之后的简单网络的标记。如果一个库所提供输入给不止一个变迁，那么库所的一个令牌可能触发其中的一个目标变迁的点火 (非确定的一个或其他点火)。

Petri 网的变迁类似于数据流角色。当有足够的输入可用时它们将点火。在基本的 Petri 网中，令牌没有值，变迁的点火行为并不涉及任何令牌的计算。点火行为仅仅扮演移动令牌从一个库所到另一个库所的角色。另外，与数据流缓冲区不同，库所不会维持令牌顺序。与同构 SDF 一样，变迁是通过每个输入库所上的令牌使能的。与 SDF 不同，一个库所可能供给了不止一个变迁，导致了模型的不确定性。

有很多 Petri 网的变种，至少有一种与 SDF 等价。特别地，令牌可以拥有值 (这些令牌在文献中被称为着色令牌 (colored token)，其中令牌的颜色代表它的值)。变迁可以操控令牌颜色 (类似于数据流的点火功能)。连接库所和变迁的弧可以被加权，这意味着点火变迁需要多个令牌，或者一个变迁产生多个令牌。这类似于 SDF 的产生和消耗速率。最后，Petri 网图结构可以被约束，这样对于每个库所，都有一个明确的源变迁和一个明确的目的变迁。这种有保序库所的 Petri 网是 SDF 图。

补充阅读：逻辑角色

这些角色可以在 Logic 库中找到，它们对建立控制逻辑很有帮助：



- **Comparator** 角色比较两个 double 类型的值 (或者任何可以无损地转换成 double 的类型，如第 14 章所解释的那样)。可用的比较操作是 >、>=、<、<= 和 ==。输出为布尔值。
- **Equals** 角色比较任意数量的任意类型输入令牌的相等性，如果它们相等，则输出一个布尔真值，否则输出假。
- **IsPresent** 角色：如果它的输入在点火时到达，则输出布尔真值，否则输出假。在数据流域，输入总是会到达，因此输出总是为真。这个角色在 SR 和 DE 域会更有用 (第 5 章和第 7 章)。

- LogicalNot 接收一个输入布尔值并输出该布尔值的相反值。
- LogicGate 实现以下 6 个逻辑功能（不同的逻辑功能对应不同的图标）：



- TrueGate 角色：当输入为布尔真值时，它产生一个布尔真值输出；否则，它不产生任何令牌。这很明显不是 SDF 角色，但它可以与 DDF 一起使用。它在 SR 中同样有用（第 5 章）。

3.2 动态数据流

虽然保证有界缓冲区和排除死锁是很有价值的，但是这是有代价的：SDF 的表达能力不够好。它不能直接表达有条件的点火行为，比如，如何让一个令牌有特定值时角色才点火。

现在已经开发出大量的可以不受 SDF 约束的数据流变体。我们在这一节中介绍一个称为**动态数据流**（Dynamic Data Flow, DDF）的变体。DDF 比 SDF 更灵活，因为角色在每次点火时可以产生和消耗不同数量的令牌。

3.2.1 点火规则

与在其他数据流 MoC（如 SDF）中一样，当 DDF 角色有足够的输入数据时，它们才开始点火。对于一个要点火的角色，在角色点火前必须遵守**点火规则**（firing rule）（即角色需要满足点火条件才能进行点火）。在 SDF 模型中，角色的点火规则是恒定的。规则明确指定了角色可以进行点火之前，其每个输入端口需要的令牌数。但是在 DDF 域中，点火规则更复杂，可能会为每次点火规定不同令牌数。

例 3.8 例 3.6 的 SampleDelay 角色直接为 DDF 计算模型，不需要任何对初始令牌的特殊处理。则 SampleDelay 的点火规则特别指出，在首次点火时不需要输入令牌；在后续的点火中，需要一个输入令牌。

另一个区别是，在 SDF 中，角色在每个输出端口都产生固定数量的令牌。而在 DDF 中，产生的令牌数量是可以变化的。

例 3.9 在首次点火时，SampleDelay 角色产生一定数量的令牌，该数量由它的 initialOutputs 参数指定。在后续的点火中，它仅产生一个令牌，这与它消耗的令牌数相等。

点火规则本身不必是恒定不变的。点火时，DDF 角色可能会改变下一次点火的点火规则。

BooleanSelect 是一个重要的 DDF 角色，它根据一布尔值控制令牌流，将两个输入流归并为一个流。该角色有 3 个点火规则。初始情况下，要求 control（底部）端口有一个令牌，其他两个端口没有令牌。当角色点火时，记录控制令牌的值并改变它的点火规则，即要求在 trueInput 端口（标有“T”）或者 falseInput 端口（标有“F”）有一个令牌，这取决于控制令牌的值。当这个角色下一次点火时，它消耗相应端口的令牌并将它发送到输出端口。这样，它点火两次产生一个输出。当产生一个输出后，它的点火规则变为要求在 control 端口有一个令牌。

BooleanSelect 角色的一个更通用的版本是 Select 角色，它根据一个使用整数值的控制令牌流，将任意数量的（而不仅限于两个）输入流合并为一个输出流。

BooleanSelect 角色和 Select 角色将输入流进行合并，而 BooleanSwitch 角色和 Switch 角色的作用正好与之相反，它们将一个单输入流分成多个。同样，对于每一个输入令牌，控制令牌流决定它将发送到哪一个输出流。这些 Switch 和 Select 角色完成对令牌的条件路由，如图 3-10 所示。

例 3.10 图 3-10 使用 BooleanSwitch 和 BooleanSelect 角色完成有条件点火，相当于命令式程序语言中的 if-then-else。在这个图中，Bernoulli 角色产生一个随机的布尔值令牌流。这个控制流控制 Ramp 角色产生的令牌的路由。当 Bernoulli 角色产生 true 时，Ramp 角色的输出使用 Scale 角色进行乘以 -1 的操作。当 Bernoulli 角色产生 false 时，就使用 Scale2，它将输入不加改变地进行传递。BooleanSelect 角色使用相同的控制流来选择合适的 Scale 输出。

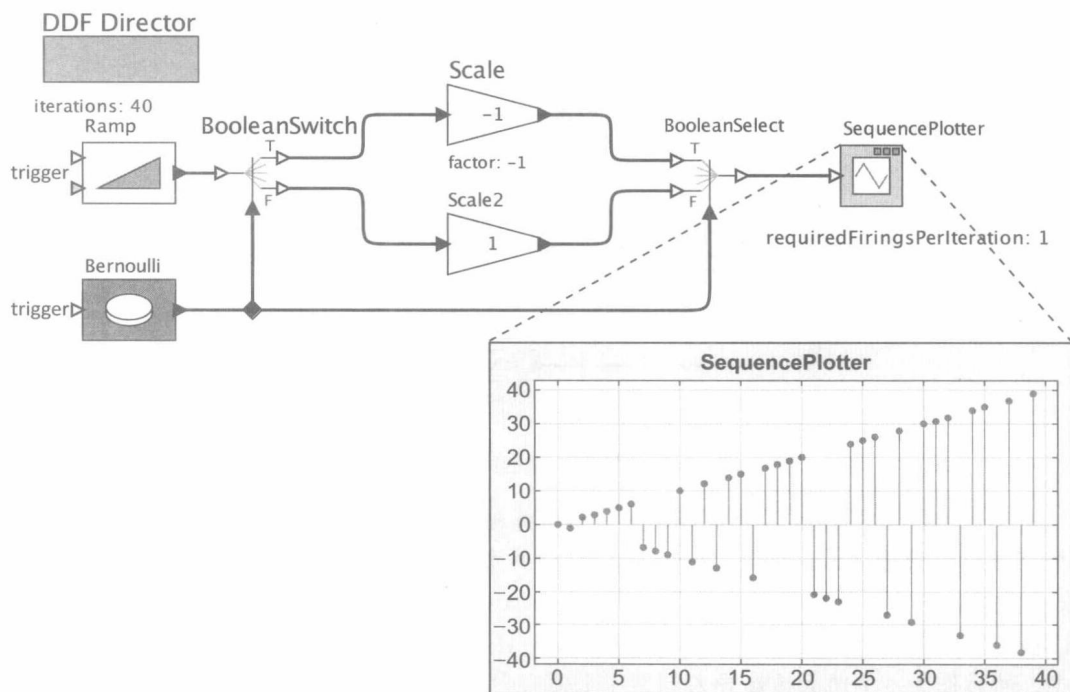


图 3-10 实现了有条件点火的一个 DDF 模型

例 3.11 如图 3-11 所示为一个 DDF 模型，它使用 BooleanSwitch 和 BooleanSelect 角色，通过一个反馈回路来实现有数据依赖的迭代。Ramp 角色用一个递增的整数序列“0, 1, 2, 3...”对回路进行反馈。SampleDelay 角色对回路进行初始化，方法是向 BooleanSelect 的 Control (控制) 端口提供一个 false 令牌。在整个循环中，每个输入的整数被反复地乘以 0.5，直到结果值小于 0.5。令牌可以是被返回到回路中以便参与另一次迭代，也可以被路由到回路外面而到达 Discard 角色 (图的右边，那个类似电路图中接地图标的符号，可在 Sinks → GenericSinks 库中找到)，这是由 Comparator 角色 (在 Logic 库中) 控制的。Discard 角色接收并丢弃它的输入，但是这种情况下，它也被用来控制一次迭代的含义。参数 requiredFiringsPerIteration 被加到该角色上，并赋值为 1 (见 3.2.2 节)。这样，模型的一次迭代包含了很多所需的循环迭代，以便产生 Discard 的一次点火。这种结构可以类比于命令式程序语言中的 do-while 循环。

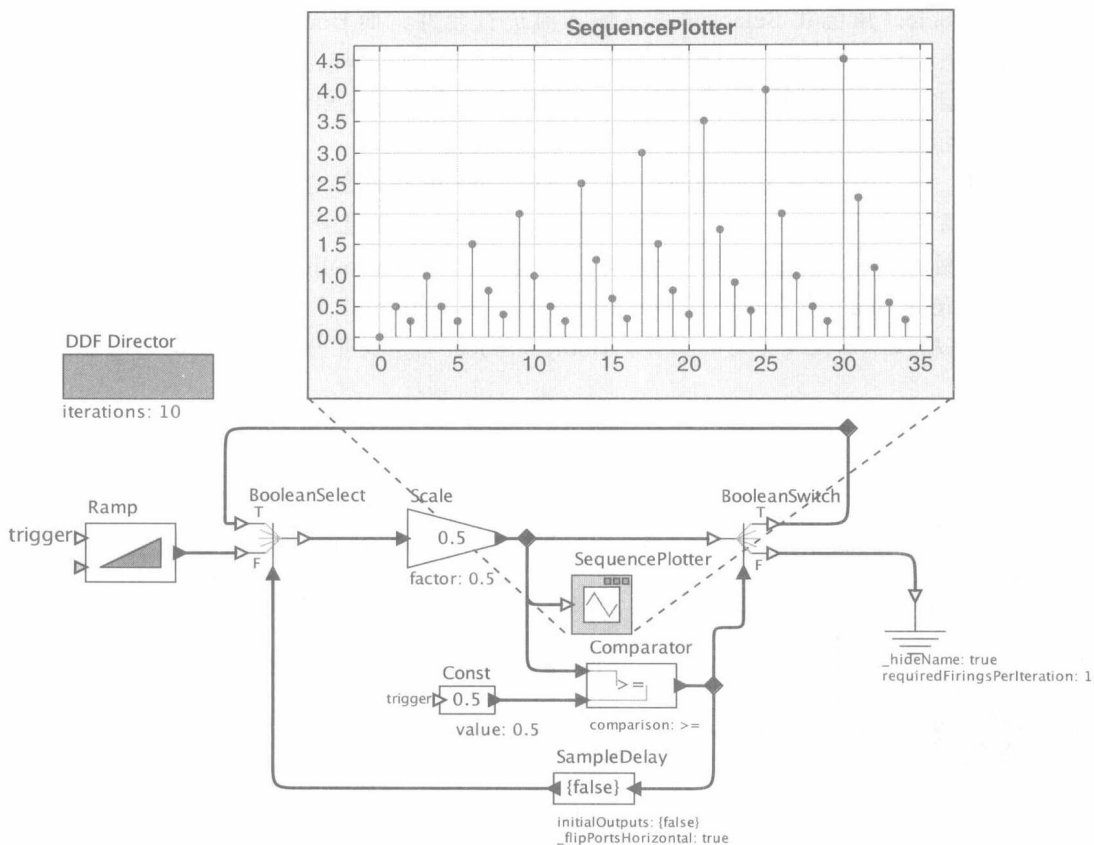


图 3-11 实现了数据依赖迭代的一个 DDF 模型

图 3-11 中所示的模式是非常有用的，它可能被反复使用。幸运的是，Ptolemy II 包括了一个可以对设计模式 (design pattern) 进行存储并重用的机制，这个机制由 Feng (2009) 提出。比如图 3-12 中所示的模式，位于 MoreLibraries → DesignPatterns 库中，它就是一个可以被重用的单元。事实上，任何一个 Ptolemy II 模型都可以作为一个设计模式导出到一个库中，然后作为一个单元重新导入到另一个模型中，导入时仅需将它拖拽到模型中。

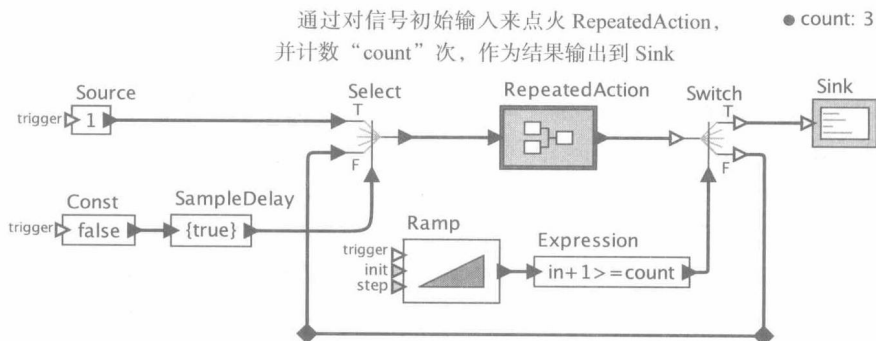


图 3-12 作为库中单元部件存储的设计模式

Switch 和 Select 角色（以及它们布尔形式的版本）是 DDF 域中的一部分，这部分相对于 SDF 域更加的灵活和具有更好的表达能力，但是使用它们也意味着有可能无法确定一个

使用有界缓冲区的调度，也不能确保该模型不会出现死锁。事实上，Buck（1993）证明，对于 DDF 模型来说，有界缓冲区和死锁是不可判定的（undecidable）。因为这个原因，导致 DDF 模型的分析变得相当困难。

例 3.12 图 3-10 中的 if-then-else 模型的一个变体如图 3-13 所示。在这个例子中，BooleanSelect 角色的输入被反转。不同于以前的模型，这个模型没有确保有界缓冲区的调度方法。Bernoulli 角色有可能产生一个任意长的值为 true 的令牌序列，在此期间，这个任意长的令牌序列为 BooleanSelect 的 false 端口建立在输入缓存区上，因此有可能发生缓冲区溢出。

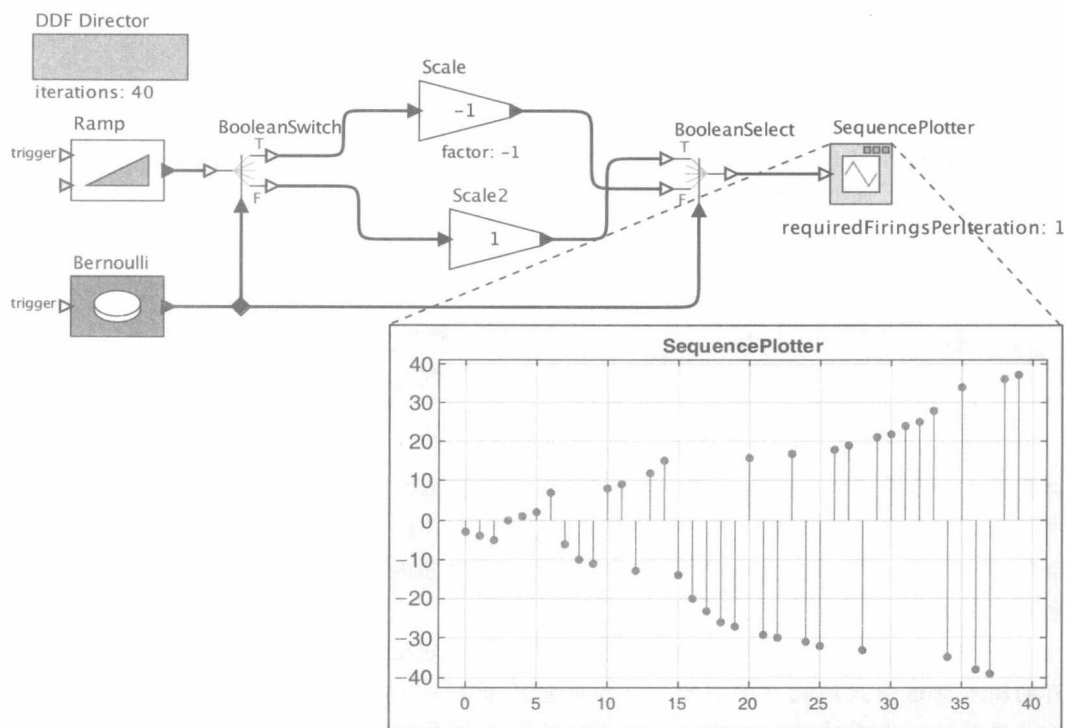


图 3-13 没有有界缓冲区调度的 DDF 模型

Switch 和 Select 角色以及它们的布尔形式，是类似于命令式程序语言中 goto 语句的数据流。它们通过对令牌进行有条件的路由以便提供对模型执行的初级控制。与 goto 语句一样，它们在模型中的使用可能导致理解困难。这个问题可以使用结构化数据流（structured dataflow）来解决，它在 Ptolemy II 中可用高阶角色来实现，这点在 2.7 节有描述。

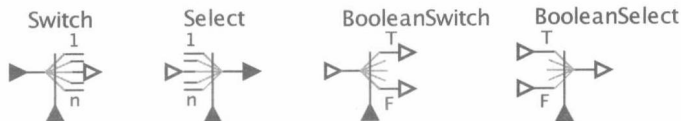
3.2.2 DDF 中的迭代

SDF 的一个优点是一个完整迭代的定义是唯一的。它由模型中每个角色固定数量的点火行为组成。因此比较容易通过设置 SDF 指示器（iteration）的参数来控制模型总体执行的持续时间，这个参数控制每个角色将被执行的次数。

DDF 指示器也有一个 iterations 参数，但是定义一次迭代不是这么简单的。可以通过以下方法定义一次迭代：向一个或多个 requiredFiringsPerIteration 角色添加参数，并给这个参数赋一个整数值，如例 3.13 所示：

补充阅读：令牌流控制的角色

Ptolemy II 提供了一些可以在模型中对令牌进行路由的角色。其中最基础的就是 Switch 和 Select 角色（以及它们的布尔形式），如下所示。



每次点火，Switch 消耗一个输入的令牌和控制（control）端口（在底部）的一个整数值令牌，并按照控制令牌的指示为输入令牌路由，以便将其传送到相应的输出通道。所有其他的输出通道在这次点火中不产生令牌。Select 则相反，它消耗一个来自于由控制令牌指定通道的令牌，并将它传送到输出。其他的输入通道不消耗令牌。BooleanSwitch（BooleanSelect）是 Switch 的变体，它的输出或输入的数量被限制为两个，并且控制令牌是布尔型而不是整数。

Switch 和 Select 可以和以下角色相比较，它们的功能是相关的：



ConfigurationSwitch 和 BooleanSwitch 相似，除了它将一个控制（control）输入端口替换为一个参数（parameter），这个参数决定了把数据传送到哪个输出。如果在模型执行期间参数值没有改变，那么这个角色就是个 SDF 角色，它总是在一个输出产生零个令牌而在另一个输出产生一个令牌。ConfigurationSelect 与 BooleanSelect 的相似之处也和这个类似。

BooleanMultiplexor 和 Multiplexor 与 BooleanSelect 和 Select 相似，但是它们从所有的输入通道中只消耗一个令牌。这些角色只保留一个输入令牌，其他的输入令牌都丢弃，并将这一个令牌传送给输出。因为这两个角色严格地在每个通道上消耗和产生一个令牌，所以它们是同构 SDF 角色。

补充阅读：结构化数据流

在一种命令式语言中，结构化程序设计使用嵌套的 for 循环、if-then-else、do-while 和递归语句替代了 goto 语句（在 Dijkstra（1968）中指出 goto 语句可能有问题）。在结构化数据流中，这样的概念同样适用于数据流建模环境。

图 3-14 展示了可以完成图 3-10 条件点火的替代方法。结果是，SDF 模型优于图 3-10 所示的 DDF 模型。与 2.7 节讨论的一样，Case 角色就是一个高阶角色的例子。其包含两个子模型（细化模型）：一个名为 true，它包含一个参数为 -1 的 Scale 角色；一个名为 default，它包含一个参数为 1 的 Scale 角色。当给 Case 角色的控制输入是 true 时，true 细化模型执行一次迭代。对于其他的控制输入执行默认细化模型。

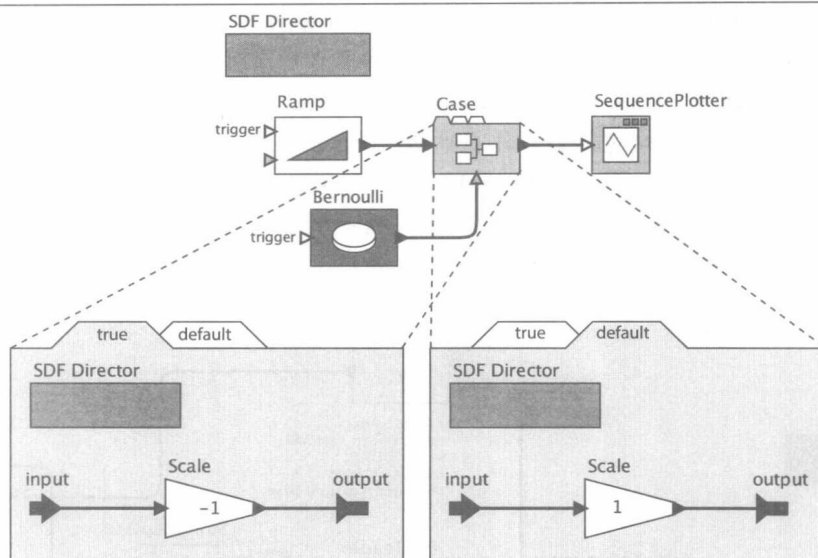


图 3-14 结构化的数据流方法用于条件点火

条件点火类型的数据流叫作**结构化数据流** (structured dataflow), 因为与许多结构化程序一样, 控制结构都是分层嵌套的。这种方法避免了有任意的数据依赖的令牌路由 (这也类似于避免了使用 goto 语句产生的任意分支), 此外, Case 角色的使用可以使所有模型都是 SDF 模型。在图 3-14 的例子中, 每一个角色在每一个端口都消耗并产生一个令牌。所以, 这个模型是否存在死锁和有界缓冲区是可分析的。

结构化数据流的类型在 LabVIEW 中有详细的介绍, LabVIEW 是 National Instruments (Kodosky et al., 1991) 开发的一个设计工具。除了提供一个和图 3-14 类似的条件操作外, LabVIEW 也为迭代 (与命令式语言中的 for 和 do-while 循环类似) 和序列 (它在一个子模型的有限集中循环) 提供结构化数据流设计。Ptolemy II 中的迭代可以使用 2.7 节的高阶角色来实现。序列 (以及更复杂的控制结构) 可以使用第 8 章所述的模态模型来实现。Ptolemy II 支持递归使用 ActorRecursion 角色, 其位于 DomainSpecific → DynamicDataflow 中 (见练习 3)。然而, 如果不加注意, 递归中的有界性又将变得不可确定了 (Lee and Parks, 1995)。

例 3.13 在例 3.10 中讨论过的图 3-10 中的 if-then-else 的例子。指示器的 iterations 参数设置为 40, 图上也确实有 40 个点。这是因为一个名为 requiredFiringsPerIteration 的参数被添加到 SequencePlotter 角色中并被赋值为 1。因此, 每次迭代都必须包括至少一次 SequencePlotter 角色的点火。这种情况下, 模型中没有其他角色有名为 requiredFiringsPerIteration 的参数, 所以该参数决定了一次迭代的内容。

当模型中的多个角色有名为 requiredFiringsPerIteration 的参数时, 或者当没有这样的参数时, 情况就更微妙。在这些情况下, DDF 仍有良好定义的迭代, 但是其定义的复杂度可能会让设计者大吃一惊。

例 3.14 再次考虑图 3-10 的 if-then-else 例子。若从 SequencePlotter 角色中删去 requiredFiringsPerIteration 参数, 那么模型的 40 次迭代将只产生 9 个输出。为什么? 回想 3.2.1 节的 BooleanSelect 角色, 对于它的每个输出, 它都要点火两次。缺少了模型中的任何

约束，在一次迭代中 DDF 指示器点火任何角色都将不会超过一次。

例 3.15 如图 3-15 所示为一个 DDF 模型，对于在一个目录下的所有 Ptolemy 模型，它将所有 SequencePlotter 角色的实例替换为 Test 角色的实例。这个模型使用 DirectoryListing 角色来为特定目录下的角色构建一个文件名数组。Ptolemy 模型查找名称为 “*.xml” 的文件形式。DirectoryListing 角色的 firingCountLimit 参数保证了这个角色只点火一次。它将在输出产生一个数组令牌，然后拒绝再次点火。一旦这个数组中的数据被处理完，就没有需要处理的令牌，所以这个模型会陷入死锁，并将停止执行。

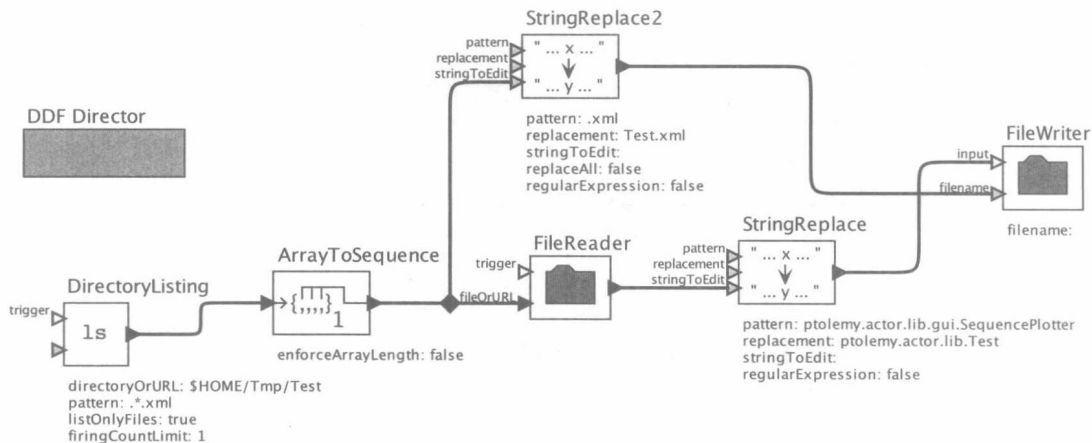


图 3-15 一个 DDF 模型，对于一个目录下的所有 Ptolemy 模型，它将所有 SequencePlotter 角色的实例均替换为 Test 角色的实例

ArrayToSequence 角色，将一个文件名数组转换为一个令牌序列，每个文件名有一个值为字符串类型的令牌。注意 enforceArrayLength 角色参数在这里被设置为 false。如果我们知道问题中 XML 文件的准确数目，我们就可以将这个参数保留为默认值，然后将数组长度参数 arrayLength 设为文件数目，并使用 SDF 指示器代替 DDF 指示器。ArrayToSequence 角色会消耗一个令牌并产生一个固定的、已知的输出令牌，因此它是一个 SDF 角色。但是，因为我们一般情况下不知道目录中会有多少匹配文件，所以还是 DDF 指示器更有用。

FileReader 角色读取 XML 文件并将其内容以字符串的形式输出。StringReplace 角色将所有的具有完整类名的 SequencePlotter 角色的实例都替换为 Test 角色的完整类名。

第二个 StringReplace 角色名为 StringReplace2，用于从原始文件名创建一个新文件名。比如说，文件名 Foo.xml 会变成 FooTest.xml，然后 FileWriter 角色将修改后的文件名写入一个具有新文件名的新文件中。

注意，可以用 IterateOverArray 角色和 SDF 指示器作为替代完成该操作（见 2.7.2 节）。我们将其留作练习，由读者自行研究（见本章末的练习 2）。

3.2.3 将 DDF 与其他域结合

虽然一个系统整体上被建模为 DDF 是最好的，但是它也可能包含一些可以被建模为 SDF 的子系统。这样，一个 DDF 模型可能包含一个具有 SDF 指示器的不透明复合角色。这种方法可以提高效率并且可以更好地控制迭代中的计算量。

反过来，如果一个 DDF 模型在它的输入/输出边界上的行为类似于 SDF，那么这个

DDF 模型也有可能和一个 SDF 模型放在一起。为了能在 SDF 模型中使用，一个不透明的 DDF 复合角色应当消耗并产生固定数目的令牌。通常由 DDF 监视器决定边界上有多少令牌是不可能的（这通常是个不确定性的问题），所以它要由模型设计者来声明消耗速率和产生速率。如果它不等于 1（不必明确声明），那么模型设计者可以在每个输入端口创建一个名为 `tokenConsumptionRate` 的参数并将它设定为一个整数值，以此来声明消耗和产生率。相似地，输出端口应该被赋予一个名为 `tokenProductionRate` 的参数。

一旦边界上的速率确定了，则应该由设计者确保在运行期间遵守这些速率。这可以使用 `requiredFiringsPerIteration` 参数来完成，如 3.2.2 节解释的那样。另外，DDF 监视器有一个 `runUntilDeadlockInOneIteration` 参数，当该参数设置为 `true` 时，就定义了一个被基本迭代反复调用直到死锁的迭代。如果使用该参数，则它将优写于模型中可能出现的 `requiredFiringsPerIteration` 参数。

DDF 符合松散角色语义（*loose actor semantics*），意味着若 DDF 指示器用在不透明的复合角色中，那么当调用它的点火（*fire*）方法时，它的状态将改变。特别地，在它们的 *fire* 方法中数据流角色会消耗（*consume*）输入令牌。一旦令牌被消耗，它们在输入缓冲区中就不再可用。这样，第二次点火就会使用新数据，而不管是否调用后点火（*postfire*）方法。由于这个原因，DDF 和 SDF 复合角色不应在要求严格角色语义（如 SR 域和连续）的域中使用，除非模型的设计者可以保证这些复合角色在 Continuous 容器的 SR 的一次迭代中点火次数不超过一次。

注意，任何 SDF 模型都可以与 DDF 指示器一起运行。但是，迭代的概念是不同的。有时候，即使在有数据依赖迭代情况下，一个 DDF 模型仍能与 SDF 指示器一起运行。图 3-14 展示了一个例子，Case 角色促成了这个组合。但是有时候，即使在使用 Switch 的情况下，使用该组合仍是可能的。SDF 调度器将假设 Switch 在每个输出通道上产生一个令牌，然后依次来构建一个调度机制。当执行这个调度时，指示器会遭遇这样的角色：指示器期望它已经做好点火准备，但是角色并没有足够的输入以供点火。它们的 *prefire*（预点火）方法返回 `false`，向指示器表明角色没有准备好点火。SDF 指示器会遵守这一点，并且会在调度中跳过这个角色。但是，这个技巧相当取巧，不推荐使用。因为它可能会导致意料之外的角色执行顺序。

补充阅读：定义 DDF 迭代

DDF 迭代（*iteration*）由基本迭代（*basic iteration*）的最小数量组成（见下文），它满足 `requiredFiringsPerIteration` 参数所要求的全部约束条件。

在一次基本迭代中，DDF 指示器将对所有被使能的（*enabled*）且不可延迟的（*non-deferable*）角色进行一次点火。一个被使能的角色是指这个角色在输入端口有足够的数，或者没有输入端口。一个可延迟的角色是指这个角色的执行可以被延迟，因为下游角色不会要求它立即执行。这种情况的出现要么是因为下游角色在连接它和可延迟角色的通道上已经有了足够的令牌，要么是因为下游角色正在等待另一个通道或端口的令牌。如果没有被使能且不可延迟的角色，那么指示器会对那些被使能且可延迟的角色进行点火，这些可延迟角色的输出通道上有满足目的角色要求的最大令牌值的下限。如果没有被使能的角色，那么已经出现死锁（*deadlock*）。Parks（1995）提出以上策略，可以保证在无限执行中，缓冲区依然是有界缓冲区（*bounded buffer*）（如果出现有界缓冲区无

限执行的情况)。

实现一次基本迭代的算法如下所示。用 E 表示被使能的角色集合, D 表示可延迟的被使能角色集合。那么一次基本(默认)迭代的组成如下, 这里 ED 表示“在 E 中且不在 D 中的元素”。

```

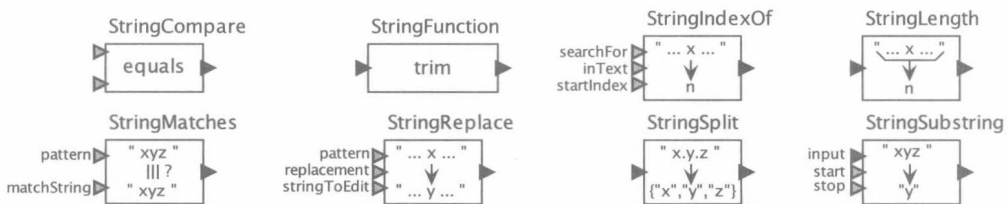
if  $E \setminus D \neq \emptyset$  then
    fire actors in  $(E \setminus D)$ 
else if  $D \neq \emptyset$  then
    fire actors in  $\text{minimax}(D)$ 
else
    declare deadlock
end if

```

函数“ $\text{minimax}(D)$ ”返回 D 的一个子集, 这个子集的元素都满足目的角色要求的输出通道上令牌数的最大值的下限。这将包括 sink 角色(无输出端口的角色)。

补充阅读：字符串操作角色

String 库提供了一些角色以便对字符串进行操作：



StringCompare 角色对两个字符串进行比较, 确定它们是否相等, 或者其中一个字符串是否是以另一个字符串开始、结尾或者包含另一个字符串。换行 StringMatches 角色检查一个字符串是否与一个给定的以普通表达式表达的模式相匹配。换行 StringFunction 函数可以删除一个字符串附近的空白空间或者将它转为大写字母或小写字母。换行 StringIndexOf 角色查找一个字符串的子串并返回这个子串的索引。换行 StringLength 角色输出一个字符串的长度。换行 StringReplace 函数将一个满足某种模式的子串替换为一个特定的替换字符串。换行 StringSplit 将一个字符串从指定分隔符处分开。换行 StringSubstring 根据给定的起始和终止索引在一个字符串中提取子串。

补充阅读：回归测试构建

当开发一个重要模型或者扩展 Ptolemy II 时, 良好的工程经验要求建立回归测试(regression test)。未来的很多变化可能会导致早期的应用程序失效, 从而造成系统行为的改变, 回归测试可以防止这些改变。幸运的是, 在 Ptolemy II 中, 创建回归测试十分容



易。在 `MoreLibraries` → `RegressionTest` 中可以找到其核心组件。

Test 角色将输入值和由 `correctValues` 参数指定的值相比较。这个角色有一个 `trainingMode` 参数，当它设置为 `true` 时，它记录它接收的输入值。因此，它的一个典型应用就是将角色放入训练模式中，执行模型，然后将角色从训练模型中取出，并且将模型保存在某个目录中。在这个目录中所有模型将作为日常测试的一部分被执行。（这就是 **Ptolemy II** 如何为自己创建出大量的回归测试的原因。）如果 **Test** 角色接收到任何不同于它记录的输入，模型就抛出异常。注意，确定性（`determinate`）模型的一个关键好处就是，它有构建这样的回归测试的能力。

NonStrictTest 也同样，除了它容忍（并忽视）缺失的输入，并且在它执行中的后点火（`postfire`）阶段测试输入，而不是在点火阶段。这对 **SR** 域和 **Continuous** 域这两个迭代终止值是固定值的域来说是很有用的。

有时候，希望模型抛出异常。对于这样的模型，回归测试应当包含一个 **TestExceptionAttribute** 的实例，**TestExceptionAttribute** 也有一个训练模式。如果模型执行中不抛出异常，或者如果抛出的异常不符合预期，模型的这种实例会使得模型抛出异常。

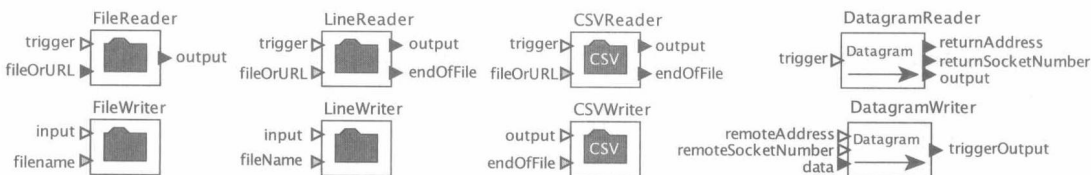
3.3 小结

数据流是一个简单和通用的计算模型，其中角色的执行由输入数据的可用性来驱动。它对流（`expressing streaming`）应用的表示特别有用，其中长数据值序列通过计算选择路径，如常见的信号处理和多媒体应用。

SDF 是一个简单的（尽管是有限制的）数据流形式，它能进行大量的静态分析和有效的执行。**DDF** 更为灵活，但是对它的控制也更具挑战性，并且执行开销更大，因为它在运行时进行调度决策的定制。它们两者可以整合到一个模型中，因于 **DDF** 产生额外的开销，因此仅应用在非用不可的地方。**SDF** 和 **DDF** 在模态模型中的良好运用将在第 8 章解释。使用 **SDF** 和 **DDF** 的模态模型提供一个通用的并发程序模型。

补充阅读：IO 角色

以下是在 **IO** 库中重要的输入 / 输出角色：



FileReader 和 **FileWriter** 通过 **URL** 或者 **URI** 从本地磁盘或从指定的远程位置读和写文件。对于 **FileReader**，在一个单个输出字符串令牌上产生文件的全部内容。对于 **FileWriter**，将每个输入字符串令牌写入一个文件，重写之前的文件内容。在这两种情况下，可以为每次点火给出新的文件名。为了从标准输入读取，指定 `System.in` 为文件名。为了写入标准输出，指定 `System.out` 为文件名。**LineReader** 和 **LineWriter** 类似，除了

它们每次读和写一行。

CSVReader 和 **CSVWriter** 读和写 CSV 格式或者逗号分隔值（分隔符可以是任何东西，不一定是逗号）的文件或 URL。将 CSV 文件转换成记录令牌，并将记录（record）令牌转换为 CSV 文件。文件的第一行定义记录的文件名。为了使用 **CSVReader**，需要帮助类型系统决定输出类型。其最简单的实现方法是可以使用后向类型推断（backward type inference）（见 14.1.4 节）。它设置 **CSVReader** 角色输出端口的数据类型为最通用的类型，这个类型能被角色下游接受。例如，如果角色下游从记录中提取字段，类型约束将自动地要求那些字段出现并具有兼容的类型。也可以强制输出类型使用 [Customize→Ports] 上下文菜单命令。

在 IO 库中还包含了下面的角色：



ArrowKeySensor 对键盘的方向键做出响应，并产生输出。

DirectoryListing 输出指定目录下与命名某一模式相匹配的文件名数组。

练习

- 3.1.3 节补充阅读：多速率数据流角色中描述的多速率角色与第 2 章补充阅读：数组处理角色和补充阅读：数组构建与拆分角色中描述的数组角色对用 SDF 构建集合操作（collective operations）很有用，这些都是对数据数组的操作。这个练习探讨利用 SDF 实现所谓的**多对多分散 / 收集**（all-to-all scatter/gather）。具体地说，就是构建一个模型，使它产生以下所示的 4 个数组，它们的值如下：

```

{"a1", "a2", "a3", "a4"}
{"b1", "b2", "b3", "b4"}
{"c1", "c2", "c3", "c4"}
{"d1", "d2", "d3", "d4"}

```

并将它们转换为具有如下值的数组

```

{"a1", "b1", "c1", "d1"}
{"a2", "b2", "c2", "d2"}
{"a3", "b3", "c3", "d3"}
{"a4", "b4", "c4", "d4"}

```

请用 **ArrayToElements** 和 **ElementsToArray**，以及 **ArrayToSequence** 和 **SequenceToArray**（对于后者，可能需要 **Commutator** 和 **Distributor**）。评论方法的优缺点。提示：可能必须明确地将连接的通道宽度设置为 1。双击线路并设置值。可能要尝试用 **MultiInstanceComposite**。

- 如图 3-15 中的模型，在例 3.15 中有讨论。使用 **IterateOverArray** 角色实现这个相同的模型，并且只用 SDF 指示器而不用 DDF 指示器（见 2.7.2 节）。
- Ptolemy II 中的 DDF 指示器支持一个称为 **ActorRecursion** 的角色，它是一个对包含它的复合角色的递归。例如，图 3-16 所示的模型实现 Eratosthenes 筛选法，其用来寻找素数的方法详见 kann 和 MacQueen（1977）。

使用这个角色实现一个计算斐波那契数的复合角色。即复合角色的一次点火需要实现以下点火函数：

$f: N \rightarrow N$, 对于所有 $n \in \mathbb{N}$,

$$f(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ f(n-1) + f(n-2) & \text{否则} \end{cases}$$

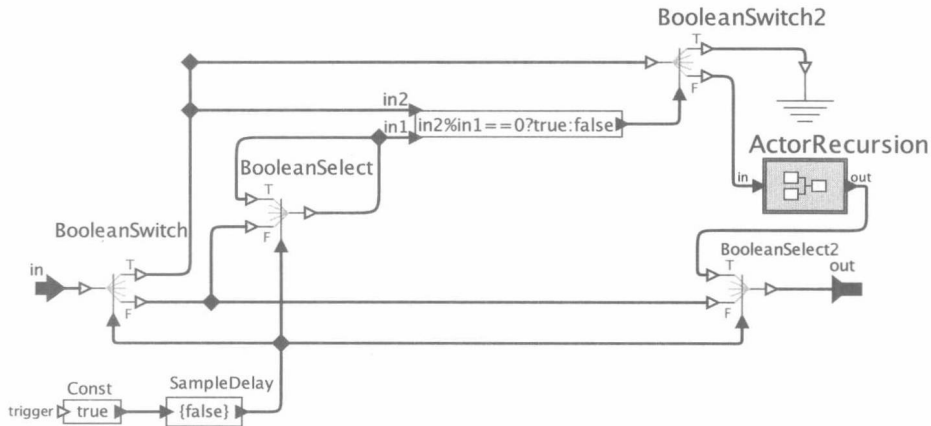


图 3-16 在 DDF 中使用 ActorRecursion 的 Eratosthenes 筛选法

当 ActorRecursion 点火时, 它复制分层结构中上层的复合角色 (即它的容器, 或者它容器的容器等), 层次的名称与它的 recursionActor 参数值相同。ActorRecursion 的实例用那些与容器匹配的端口来填充。这个角色应该视为对特殊种类高阶角色的一个高度的实验实现 (experimental realization)。这是一个高阶角色, 因为它的参数由一个包含它的角色来指定。然而, 它的实现是非常低效的。它每次点火所引用的角色的副本会带来巨大的时间和空间开销。一个更好的实现方式是使用类似于面向过程语言中所使用的栈帧的方法 (stack frame approach)。而它使用的这个方法更像是在运行时复制源代码并解释它。为了提高执行效率, 如果角色已经提前创建, 那么要避免创建副本。仅凭图 3-16 中的图像, 没有办法分辨 ActorRecursion 实例引用了哪个复合角色。因此, 无法从它的可视化表示中真正地阅读该程序。

进程网络和会话

Neil Smyth、John S. Davis II、Thomas Huining Feng、Mudit Goel、
Edward A. Lee、Thomas M. Parks 和 Yang Zhao

计算的数据流模型是并发的。点火顺序只受数据优先级的约束，所以指示器对点火顺序的确定有很大的自由度。两个彼此不依赖的角色可以同时点火。事实上，这是调度器长久以来的做法。（详见第3章补充阅读：SDF 调度器）不过，第3章中描述的指示器每次只点火一个角色。本章介绍了两种指示器，它们可以将角色并发地点火。这两种指示器在语义上与数据流比较相似，但用途却不同。

本章中的指示器使每个角色在自己的线程上执行。**线程（thread）**是一个可以与其他并发执行的线程共享变量的顺序程序。在多核机器上，两个线程可以在不同的核上并行地执行。在单核机器上，每个线程的指令可以被任意地交叉执行。指令的任意交叉执行使得理解线程之间的交互变得十分困难（Lee, 2006）。本章描述的指示器为线程的交互提供了一种更容易理解和预测的途径。

使用本章中指示器的一个表面上的动机就是可以更好地利用多核系统的并行性。例如前几章中描述的模型，也可以由一个 PN 指示器来执行（详情如下所述），角色在多核上将同时点火。也许模型这样执行会比在 SDF 指示器下执行要快。但是，实际与之相反，不管核的数量是多少，PN 模型几乎总是比 SDF 模型运行得慢。这种情况的原因可能是：与互斥锁（mutual exclusion lock）相关的线程交互的开销，使得 Ptolemy II 模型在运行时被编辑。原则上，这个功能应是禁用的，性能的提升是有希望的。但是在撰写本书时，Ptolemy II 中还没有这样的机制。因此，在 SDF 或 DDF 模型适用的情形下，采用本系统的线程指示器以超越它们是无充足原由的。

在本章中，我们将重点放在这类模型上，这些模型的目标要求角色进行并发点火。这些例子展示了此类指示器的确在表达方式上具有根本性的进步，而不仅仅是性能方面有所提升。

4.1 Kahn 进程网络

Kahn 进程网络（不同的称法有 KPN、进程网络（process network）或者 PN）是与数据流模型紧密相关的计算模型，由提出者 Gilles Kahn 的名字命名（Kahn, 1974）。数据流和 PN 的关系在 Lee and Parks（1995）和 Lee and Matsikoudis（2009）有详细的研究，但是都是非常简单的短文。在 PN 中，每个角色在它们自己的线程中并发地执行。即一个 PN 角色不是由它的点火规则和点火函数来定义，而是由一个（通常是无终止的）程序来定义的，该程序从输入端口读取令牌并向输出端口写令牌，所有角色同时执行（至少在概念上是，将它们是否同时执行或语义是否交叉都是无关紧要的）。

刚开始，Kahn（1974）给出了非常简洁的数学条件来确保这些角色组成的网络是确定的（determinate）。在这种情况下，“确定的”意味着在角色之间的每个连接上的令牌序列都

是唯一定义的，尤其是对于进程的调度来说也是唯一的。每一个合法的线程调度明确地产生相同的数据流。因此，Kahn 说明了在没有非确定性（nondeterminism）的情况下并发执行是可能的。

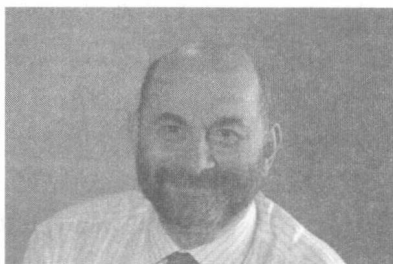
三年后，Kahn and MacQueen（1977）给出了一个简单又易于程序实现的机制，该机制确保满足数学条件即保证了确定性。机制的关键部分在于，不论何时只要进程读输入数据，在输入端上都会执行阻塞读不用该（blocking read）。具体地说，阻塞读意味着，如果进程选择从一个输入端口访问数据，它将发出一个读请求并阻塞，直到数据可用。它无法做到测试输入端口以知晓数据是否可用，然后执行基于数据是否可用的条件分支，因为这样的分支会引起调度依赖（schedule-dependent）行为。

读锁定和点火规则密切相关。点火规则指定继续计算（随着一个新的点火函数）所需要的令牌。类似地，阻塞读指定继续计算（通过进程的继续执行）所需要的一个单令牌。Kahn 和 MacQueen 表明，如果每一个从输入序列到输出序列的角色都实现一个数学函数（意味着对每个输入序列、输出序列都是唯一定义），那么阻塞读将足够保证确定性。

当一个进程写一个输出端口时，它执行非阻塞写（nonblocking write），意味着写立即执行和返回。进程不会被阻塞以等待接收进程准备好接收数据^①。这就是如何写输出端口，MoC 在数据流中工作也如此。因此，数据流和 PN 之间唯一本质上的区别是，PN 的角色不会在点火函数中发生故障。它的目标是作为一个不断执行的程序。角色的点火不需要是有限的。

历史笔记：进程网络

通过消息传递进行并发进程交互的概念植根于 Conway 的协同程序（coroutine）（Conway, 1963）。Conway 表示，软件模块与之间的交互就如同它们在执行 I/O 操作。用 Conway 的话来说，“当协同程序 A 和 B 相互连接以便 A 发送项目给 B 时，B 持续执行直到其遇到了读命令，这意味着它需要一些来自于 A 的东西。因此 B 被中断；A 执行，A，直到 A 进行写操作，最后返回 B 的断点。”



Gilles Kahn (1946—2006)

最少的定点语义要归功于 Kahn（1974），他开发了如 CPO（完全偏序）上的连续函数进程模型。Kahn and MacQueen（1977）使用无写锁定和读锁定定义进程交互，作为一个特殊的连续函数的情形，并且开发一个定义交互程序的程序语言。他们的语言包括递归结构、一个可选函数符号和动态进程实例化。他们产生一个命令驱动执行语义，与 Lisp 语言的惰性计算（lazy evaluator）相关（Friedman and Wise, 1976；Morris and Henderson, 1976）。Berry（1976）推广了这些有稳定函数的程序。

无界列表作为数据结构的概念第一次出现在 Landin（1965）中。这是以进程网络中的进程之间的通信机制为基础的。最初来源于 Ritchie and Thompson 创作的 UNIX 操作系统（1974），包含了管道（pipe）的概念，并实现了进程网络的一种形式（仅管线）。之后，命名提供更一般形式的管道。

① 将在本章后面讨论的会话指示器，恰好不同于这一点，会话指示器在该情况下，直到与输入端进行通信的角色准备好读，写输出端才会成功。

Kahn (1974, 反阐述即并未证明 Kahn 原则 (最大程度上公平执行进程网络所需的最小固定点数), 随着渐渐被人熟知, 后来被 Faustini (1982) 和 Stark (1995) 等证明。

补充阅读：进程网络和数据流

在文献中出现的 3 种主要的数据流变种有：Dennis 数据流 (Dennis, 1974)、Kahn 进程网络 (KPN) (Kahn, 1974) 和数据流同步语言 (Benveniste et al., 1994)。前两个是密切相关的, 而第三个是完全不同的。本章讲述 Kahn 进程网络, Dennis 数据流在第 3 章讲述, 数据流同步语言将在第 5 章讲述。

在 Dennis 数据流 (Dennis dataflow) 中, 角色的行为由原子点火序列给定, 该原子点火由有效的输入数据使能。KPN 与之不同的是, 它没有原子点火的概念。一个角色是同其他角色异步和并发执行的过程。如果定义一个点火行为为发生在访问输入间的计算, 那么 Dennis 数据流可以被视为一个特殊的 KPN (Lee and Parks, 1995)。但是两者在角色和模型的设计风格上是完全不同的。Dennis 的方法是基于一个操作的概念, 该概念是由是否满足点火规则来驱动原子点火的。Kahn 的方法是一个基于进程标志的概念, 如无限流中的连续函数。

Dennis 的方法影响计算机体系结构 (Arvind et al., 1991; Srinivasan 1989)、编译器设计, 以及并发程序语言 (Johnson et al., 2004)。Kahn 的方法影响进程代数 (如 Broy and Stefanescu (2001)) 和并发语义 (如 Brock and Ackerman (1981) 和 Matthews (1995))。它在流语言 (Stephens, 1997) 和操作系统 (如 UNIX 管道) 中有实际实现。有趣的是, 上述计算模型已经随着多核体系结构推动并行计算的复苏而发展。并行程度数据流计算模型及其改进执行策略和标准被致力于机器 (Thies et al., 2002)、分布式系统 (Lzaro Cuadrado et al., 2007; Olson and Evans, 2005; Parks and Roberts, 2003) 和嵌入式系统的开发 (Lin et al., 2006; Jantsch and Sander, 2005) 及其改进执行策略 (Thies et al., 2005; Geilen and Basten, 2003; Turjan et al., 2003; Lee and Parks, 1995) 和标准 (Object Management Group (OMG), 2007; Hsu et al. 2004)。

Lee and Matsikoudis (2009) 缩小了 Dennis 和 Kahn 之间的差距, 将 Kahn 的方法如何自然地扩展到 Dennis 数据流, 包含点火的概念。通过建立点火函数和 Kahn 进程之间的关系实现类似点火序列的函数。分析的结果是点火规则和点火函数的一个形式化特征, 该特征保持模型的确定性。

Kahn and MacQueen (1977) 因为一个有趣的原因访问了 PN 网络的协同程序 (coroutines) 中的一个进程。程序或者子程序是一个被另一个程序“调用”(called) 的程序段。子程序在调用之前可完成执行, 以确保调用它的程序可以连续执行。在 PN 模型中进程之间的交互更对称, 因为没有调用者和被调用者。当进程执行读锁定时, 在某种意义上它调用了提供数据的上游进程中的程序。同样, 当执行写操作时, 在某种意义上它调用了处理数据的下游进程中的程序。但是数据的生产者和消费者之间的关系比子程序更对称。

当与 PN 指示器 (同用户定义角色相对应) 一起使用传统的 Ptolemy II 角色时, 角色在无限循环中会自动封装, 该角色被不断点火直到模型停止 (见 4.1.2 节) 或者角色终止 (通过从它的后点火 (postfire) 方法中返回 false)。当角色接收输入时, 它一直被到输入可用。当

它发送输出时，输出令牌进入一个（概念上的）无界的 FIFO 队列。令牌将最终交付给目的角色。

一个有趣又微妙的现象就是使用中的角色会因为输入端口的输入令牌是否可用而导致不同行为。例如，AddSubtract 角色将加上它的 plus 端口上的所有可用令牌，并减去它的 minus 端口上的所有可用令牌。当在 PN 指示器下执行时，该角色直到它从每个输入通道接收一个令牌才会完成它的操作。当角色询问在输入上是否令牌可用时（使用 hasToken 方法），应答总是 yes！然后当它读令牌（使用输入端口的 get 方法）时，它将阻塞直到确实有令牌可用。

与数据流一样，PN 同样存在关于缓冲区的有界性和死锁这类挑战性问题。PN 足够地说明是不可判定的（undecidable）。Parks（1995）提供了一个有界性问题的有效解决方法，由 Geilen 和 Basten（2003）阐述。Parks 提供的解决方法是 Ptolemy II 中的一种具体实现。

4.1.1 并发点火

当有在激活时没有立即返回的角色时，PN 模型会特别有用。例如，InteractiveShell 角色（在 Sources → SequenceSources 中找到），打开一个窗口（如图 4-1 中顶部和底部窗口），用户可以在窗口中输入信息。当角色点火时，不论输入是否已经接收，它都会显示在窗口中，紧随其后的是一个提示符（默认是“>>”）。然后它等待用户输入新的值。当用户输入一个值并单击输入返回时，角色输出一个包含该值的事件。



图 4-1 有两个 InteractiveShell 实例的模型，每一个实例都会阻塞以等待用户输入

当响应输入的角色点火时，直到用户在打开的窗口中完成输入，点火才完成。假如使用数据流指示器（或者任何其他每次点火一个角色的指示器），那么整个模型将会阻塞以便等待用户从一个 InteractiveShell 中输入，没有其他的角色可以点火。对于某些模型，这可能是个需要解决的问题。

例 4.1 如图 4-1 中所示的模型。该模型模拟了一种有两个用户的对话框，在继续进行其他一些动作之前，会在两个用户中寻找并发性。在这种情况下，它仅仅询问每个用户“你准备好了吗？”等待每一个用户的一个回答，如果两个用户都回答“是”，那么模型会响应“让我们进行！”在这个 PN 模型中，每个 InteractiveShell 实例都执行独立的线程。如果执行模型而不使用 SDF 指示器，那么 SDF 指示器会在一开始就点火一个，直到第一个用户响应完才会询问第二个用户问题。

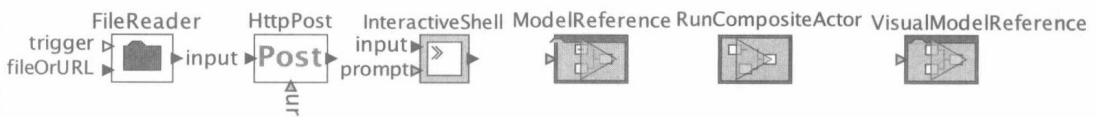
上例中，InteractiveShell 角色与 PN 模型外的世界交互。它可以通过特别配置来读写 IO，也可锁定 IO，等待外部环境的动作。实际为一个 I/O 操作，当外部动作的 I/O 操作完成之前该角色被阻塞（RP 读阻塞）。在练习 5 中的例子同样具有该属性，角色可以从传感器收集数据。

如上文所述，Kahn 进程网络的关键属性是模型具有确定性（deterministic）。然而，宣称图 4-1 中的模型是确定性的可能看起来很奇怪。然而，清楚的是通过 Expression 角色产生的值序列不同于从执行到执行，因为它取决于打开的对话框窗口中的用户输入。事实上，确定性需要每个角色从输入序列到输出序列实现一个数学函数。严格地讲，InteractiveShell 没有实现这样的函数。在两次不同的执行中，相同的输入序列不会产生相同的输出序列，因为输出取决于用户的输入。尽管如此，Kahn 的这一属性是有意义的，因为它断言如果确定用户的行为，那么模型的行为是唯一的且是良好定义的。如果在两次连续运行中，两个用户精确输入相同序列的响应，那么模型会有相同方式的行为。模型的行为仅仅取决于用户的行为，不取决于线程调度器的行为。

在 Ptolem II 中，可通过一个称为 NondeterministicMerge（详见本节后面的补充阅读：对 PN 模型有用的角色）的特殊角色来建立非确定性（nondeterminate）的进程网络。这样的模型会特别有效。

补充阅读：无界点火角色

下面几个角色特别受益于 PN 指示器的使用，因为它们点火时不会（必要地）立即返回：



- **FileReader** 读取本地计算机或者因特网中的 URL（统一资源定位符）上的文件。在后一种情况下，在服务响应时该角色可能会阻塞一个不确定的时间。
- **HttpPost** 给因特网上的 URL 发送一个记录，当用户以在线形式填写表单时模拟浏览器的典型工作。在服务响应时该角色可能会阻塞一个不确定的时间。
- **InteractiveShell** 打开窗口并等待用户输入。

- ModeReference 执行一个引用模型直到结束（如果模型没有终止，将一直执行）。
- RunCompositeActor 执行一个包含模型直到结束（如果模型没有终止，将一直执行）。
- VisualModelReference 打开 Vergil 窗口显示一个引用模型并执行模型直到结束（如果模型没有终止，将一直执行）。

例 4.2 如图 4-2 中所示的例子。该模型模拟了两用户之间的对话，在此过程中用户可根据任意顺序提供输入内容。在该模型中，用户通过 InteractiveShell 角色键入他们的输入到打开的窗口中，输入将由 Display 角色合并后显示。

在这个模型中，如果在两次执行过程中，用户的输入相同，结果显示却不会相同。事实上，如果两个用户接近同时键入输入，那么它们的文本在 NondeterministicMerge 的输出顺序是没有定义的，任何一个顺序都是正确的。顺序的结果将取决于线程调度器，而不是用户的键入。这与图 4-1 中的模型有明显区别。

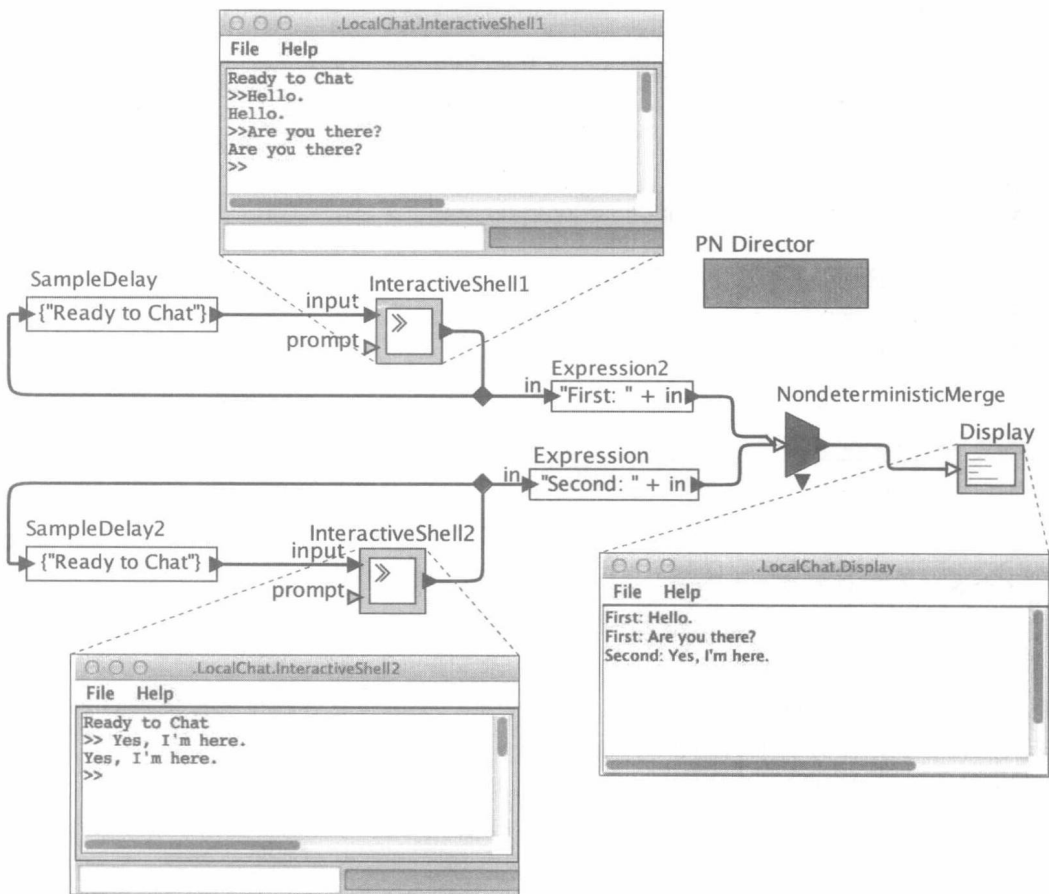


图 4-2 使用 NondeterministicMerge 创建非确定性进程网络的模型

在前面模型中引入的非确定性非常有用，但它是有代价的。例如，这样的模型非常难以测试。它没有单一的正确执行结果。由于该原因，需谨慎使用 NondeterministicMerge，并且只能在它确实需要的时候使用。如果模型可以用确定性机制来建立，那么它就应该使用确定性机制来建立。

尽管前面的例子模拟了两个客户端之间的对话，但它并不是一个真实实现的聊天应用。在同一个屏幕上打开并在同一个过程中运行两个交流窗口，因此两个不同用户几乎不能聊天。使用 PN，并使用确定性机制也是。也可以建立一个聊天客户端。

例 4.3 图 4-3 中的模型实现了一个聊天客户端。该模型假设服务器已经部署在由模型的 URL 参数指定的 URL 上。在第 16 章的练习 1 中实现了这样一个服务器。

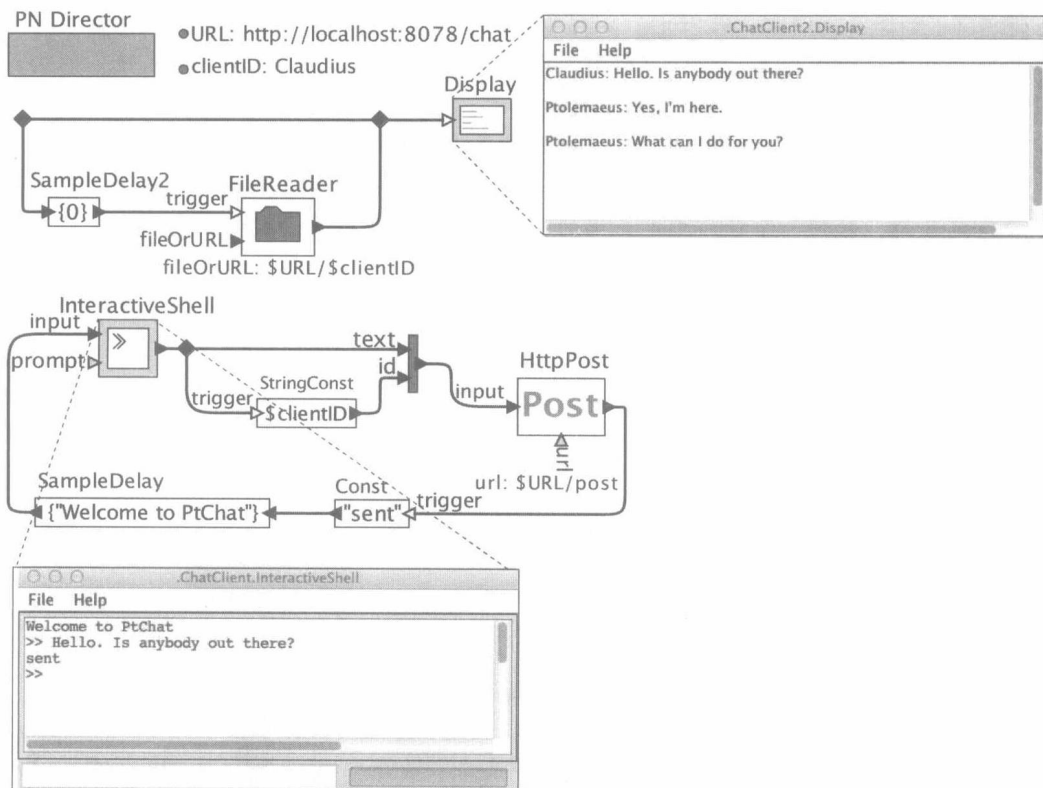


图 4-3 实现了一个聊天客户端的模型

这个客户端假设服务器提供两个接口。HTTP Get 请求用来检索聊天文本，这些文本包括这个模型的用户输入文本以及会话中任何其他参与者提供的文本。图 4-3 中的反馈回路使用 FileReader 角色产生 HTTP Get。通常，服务器不会立即响应，而是会等待直到有聊天数据到达。因此，FileReader 角色将阻塞，等待响应。收到响应后，模型将使用 Display 角色显示该响应，并产生另一个 HTTP Get 请求。

Get 请求阻塞直到所需数据到达，该种技术称为长轮询 (long polling)。实际上，它提供了一种简单的推送技术 (push technology) 的形式，其中当有数据提供时，服务器推送数据给客户端。当然，如果客户端已经产生了一个 Get 请求，并且客户端可以耐心等待响应，那么它就会工作，而不是提示超时。

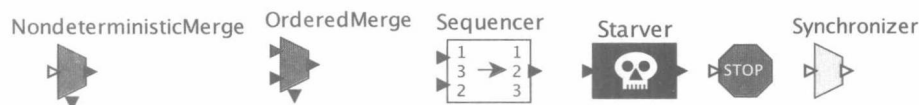
图 4-3 下面的反馈回路用来为客户端提供给聊天会话的聊天文本。它使用 InteractiveShell 角色提供用户可以输入文本的窗口。当用户提供文本时，它聚合一条包含文本和用户 ID 的记录，并使用 HttpPost 角色把这个记录传送给服务器。当服务器回复时，模型给用户提供一个“已发送”的确认，并等待用户输入新文本。

同时，服务器将通过给所有的 HTTP Get 挂起的实例回复来正常地响应 HTTP Post。在图 4-3 所示例子执行中，本地用户 Claudius 首先输入“Hello.Is anybody there?”在因特网其他地方的其他用户 Ptolemeaus 回答“Yes, I’m here.”远程用户然后进一步输入“What can I do for you?”使用长轮询技术把这个文本“推送”给聊天客户端，并因此显示在屏幕上。

前面的例子显示了使用 PN 创建两个可同时执行任务的实例，每一个都可以阻塞。该模型是确定性的，因为对于由 FileReader 和 InteractiveShell 产生的任意输出序列，模型中的所有信号都是独立定义的。因此，与图 4-2 中的例子不同，模型的行为仅仅取决于用户的输入，而不取决于线程调度。这个确定性的形式非常有效，其中很大的好处就是模型可以通过输入序列定义和响应检查来进行测试。

补充阅读：对 PN 模型有用的角色

有些角色在 PN 模型中特别有用，如下所示。



所有的这些角色可以在 DomainSpecific → ProcessNetworks 中找到，但是它们中的一些也出现在 Actors → FlowControl 中。

- **NondeterministicMerge**。通过在单个的流上任意地交错其令牌来合并令牌序列。
- **OrderedMerge**。将两个单调递增的令牌序列合并为一个单调递增的令牌序列。
- **Sequencer**。根据序列号获得输入令牌流和序列号流，并根据序列号重新排序输入令牌。在每一次迭代中，这个角色读取一个输入令牌和一个序列号。这个序列号是从零开始的整数。如果序列号是序列中的下一个值，那么将输入端口读取的令牌放在输出端口上。否则，保存它，直到它的序列号是序列中的下一个值。
- **Starver**。通过输入令牌不会改变输出，直到特定数量的令牌通过。那时，消耗和丢弃所有后续的输入令牌。这可以用于反馈回路来限制执行有限数据集。
- **Stop**。当在任何输入通道上接收到一个 true 令牌时，模型停止执行。
- **Synchronizer**。同步多个流以便它们在相同速率下产生令牌。即，当至少有一个新令牌存在于每一个输入通道时，每个输入通道消费只消耗一个令牌，并且该令牌输出到相应的输出通道。

4.1.2 PN 模型的执行停止

Ptolemy II 中的 PN 模型将一直执行，直到以下情况之一发生：

- 所有的角色都已终止。当角色的后点火（postfire）方法返回 false 时，该角色终止。Ptolemy II 中的很多角色有一个 firingCountLimit 的参数（例如，Const 角色）。设置这个参数为正整数将导致角色的进程在角色点火特定次数后终止。
- 所有的进程在读取输入端口被阻塞。这是一个类似于发生在数据流模型中的死锁。死锁可能导致模型中的错误，或者它可能是有意行为。例如，上面提到的 firingCountLimit 参数和 Starver 角色可以用于创建一个饥饿条件（starvation condition），

其中没有终止的角色将永远阻塞在永远不会被满足的读上。尽管采用“死锁”和“饥饿”两个不好的字眼，但这些都是终止 PN 模型执行完全合理和有用技术。

- Stop 角色从它的一个输入端口读取一个真值输入令牌。一旦读取这些输入，Stop 角色将与指示器协作终止所有线程的执行。具体地说，任何线程在读或写端口上被阻塞，它们将立即终止，并且任何没有被阻塞的线程将在它下一次尝试执行读或写时终止。Stop 角色可以在 `Actors` → `FlowControl` → `ExecutionControl` 库中找到。
- 缓冲区溢出。当通信通道上未消费的令牌数量大于指示器的 `maximumQueueCapacity` 的参数值时，将发生缓冲区溢出。注意，如果设置 `maximumQueueCapacity` 为 0，那么直到操作系统拒绝 Ptolemy 系统的附加存储器时才会产生溢出，在系统存储器耗尽时通常才会产生溢出。
- 在某些角色进程中发生异常。其他线程的终止的方法类似于 Stop 角色所做的。
- 在 Vergil 中，用户按 Stop（停止）按钮。该线程终止的方法类似于 Stop 角色所做的。

这些只是终止执行的机制。关于如何使用它们将在练习 6 中继续讨论和研究。

即便如此，该机制也有局限性。例如，图 4-3 中使用的 `FileReader` 角色，如果发生读阻塞，它将不能被中断。因此，终止图 4-3 中的模型是非常困难的。如果按下 Stop（停止）按钮，它最终会超时，但等待的代价可能是巨大的。这些局限是因该角色使底层的 `java.net.URLConnection` 类。

4.2 会话

与 PN 一样，在 Ptolemy II 中的会话域中，每个角色都在独立的线程上执行。与 PN 不同，角色之间的通信是通过会话（Rendezvous）而不是通过无界 FIFO 队列来实现的。准确地说，当角色通过输出端口准备传送消息时，它将锁定直到接收角色准备接收它。同样，如果角色通过输入端口准备接收消息，它将锁定直到发送角色准备好发送。因此，这个域既实现了写锁定（blocking write）又实现了读锁定（blocking read）。

这个域支持条件会话和多路会话。在条件会话（conditional rendezvous）中，一个角色将根据条件选择多个角色中的一个进行会话^①。在多路会话（multiway rendezvous）中，一个角色需要在同一时间与多个其他角色进行会话^②。一般来说，当使用条件会话时，选择哪个进行会话是不确定的，因为会话的发生通常依赖线程调度器。

会话域是基于 Hoare（1978）首先提出的通信顺序进程（CSP）模型和由 Milner（1980）提出的通信系统演算（CCS）发展而来的。CCS 也形成了 Occam 程序语言的基础（Gallety, 1996），该语言在 20 世纪 80 年代到 90 年代的这段时期为并行计算机的程序设计提供了一些成功的基础。

基于会话的通信也称为同步消息传递（synchronous message passing），但为避免与 SR（synchronous-reactive，同步响应）域混淆，SR 部分的内容将在第 5 章中描述，而 SDF 域已在第 3 章中介绍过了。

① 对于那些熟悉 Ada 语言的人来说，这类似于选择语句。

② 多路会话也称为障碍同步（barrier synchronization），因为它阻塞每一个参与的线程，直到所有参与的线程都到达障碍点，由经过一个端口发送或接收来表示。

补充阅读：对会话模型有用的角色

在会话模型中的一些特别有用的角色显示如下：



- **Barrier**：一个障碍同步（barrier synchronization）角色。该角色只有在所有输入通道上的发送角色都准备好发送时，才允许输入。
- **Buffer**：一个 FIFO 缓冲区。该角色缓冲输入提供的的数据，在需要时将它传送给输出。不论缓冲区是否为空，该角色希望和输出会话。只要缓冲区未满，它都希望与输入会话。输入以 FIFO（先进先出）的顺序传递到输出。可以为缓冲区指定有界或无界的容量。
- **Merge**：一个条件会话（conditional rendezvous）。该角色合并任意数量的输入序列到一个输出序列。它与任意输入进行会话。在收到一个输入后，将与输出会话。在成功传送输入令牌给输出之后，它再一次返回到可与任意数量输入会话的状态。
- **ResourcePool**：一个资源竞争管理器。该角色管理一个资源池，其中每个资源由一个任意值的令牌表示。在 grant 输出端口上授予资源，并将资源释放到 release 输入端口。这些端口都是多端口，因此可将资源在输出端口授予给使用该资源的多个用户，输入可由多个角色释放。初始资源池由 initialPool 参数提供，该参数是任意类型的数组。在执行时间内，该角色都准备与连接它的 release 输入端口的其他角色进行会话。在会话发生时，输入提供的令牌会增加到资源池。另外，不论资源池是否为非空，该角色都准备与连接它的 grant 输出端口的角色进行会话。如果有多个这样的角色，便是一个条件会话，而且可能导致模型的非确定性（nondeterminism）。当这类输出会话发生时，该角色将资源池中的第一个令牌传送给该输出端口，并从资源池中移除该令牌。

4.2.2 条件会话

在条件会话（conditional rendezvous）中，角色希望与其他多个角色中的一个进行会话。通常，这将导致模型的非确定性（nondeterministic）。

例 4.6 图 4-6 中的模型用于对条件会话进行了描述。该模型使用 Merge 角色（见第 4 章补充阅读）。上方的 Ramp 角色将在它的输出产生序列 0, 1, 2, 3, 4, 5, 6, 7。下方的 Ramp 将产生序列 -1, -2, -3, -4, -5, -6, -7, -8。Display 角色将对两个序列的一非确定性合并序列进行显示。

图 4-6 中的例子包括了一个带值为 true 的参数 SuppressDeadlockReporting。Ramp 角色通过指定一个有限的 firingCountLimit 来使模型“熄火”，这是一个类似于使用 PN 时的停止条件（见 4.1.2 节）。在默认情况下，指示器将报出死锁，但是由于 SuppressDeadlockReporting 参数，它会静静地停止模型的执行。该参数表明，死锁是一个正

常的终止而不是一个错误状态。

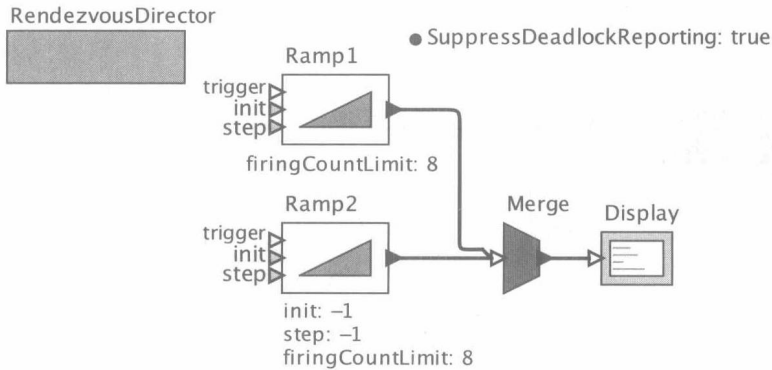


图 4-6 非确定性合并的条件交互

与上例不同，如果希望两个 **Ramp** 角色以确定的方式的输出，以便获得序列 0, -1, 1, -2, 2..., 有一种方法可以实现，该方法由 Arbab (2006) 提出，如图 4-7 所示。这个模型依赖于 **Buffer** 角色（见第 4 章补充阅读）的输入参与 **Ramp** 的实例和 **Merge** 角色的上方通道的多路会话。因为 **Buffer** 角色的容量为 1，所以在它给 **Merge** 的下方通道提供输入前，将强制该会话发生，并阻塞随后的会话直到在它下方输入提供给 **Merge**。

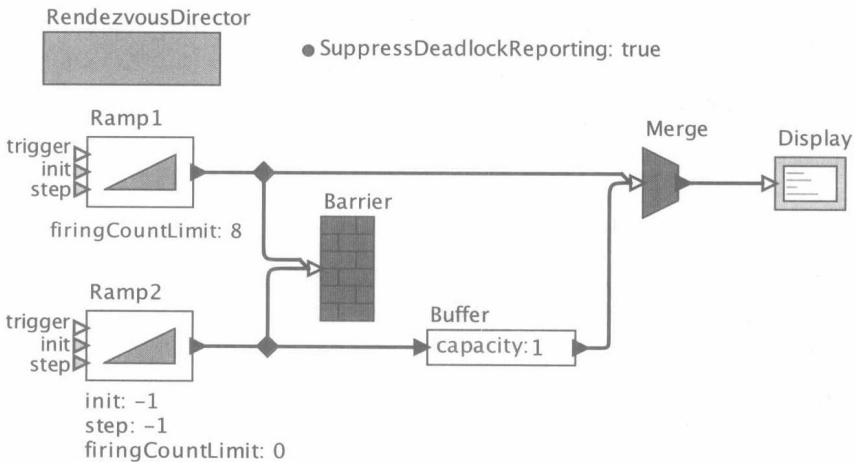


图 4-7 用来创建非确定性合并的条件交互

该模型非常巧妙，它通过使用非确定性机制来实现确定性目标。但是，构造一个更简单可实现相同目标却不需要任何非确定性机制的模型也是非常简单的（参见练习 7）。

4.2.3 资源管理

由会话指示器提供的条件会话机制非常适合角色竞争共享资源情况下的资源管理问题。一般来说，对这类模型的非确定性是可接受和期待的。**ResourcePool** 角色（见第 4 章补充阅读：对 **Rendezvous** 模型有用的角色）是这类应用的理想选择。

例 4.7 如图 4-8 中的模型对资源管理进行了说明，资源池（在图中，仅包含一个资源）可将资源非确定地提供给两个 **Sleep** 角色中的一个。该资源由一个初始值为 0 的整数表

示。每次资源被使用时，这个值将加 1。获得资源的 Sleep 角色持有一个固定的时间（分别是 100ms 和 150ms）。在这个时间以后，将释放资源，传送给一 Expression 角色，该角色使资源的值加 1 并将它返回到资源池中。

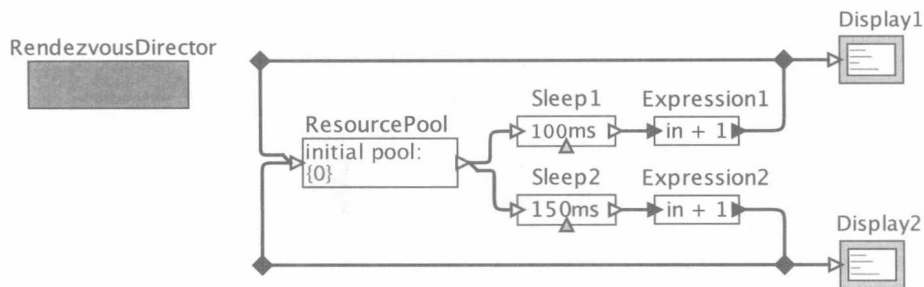


图 4-8 资源管理的条件会话

ResourcePool 角色的输入和输出端口都将实现一个条件会话。因此，具有非确定性的，当两者都准备好时，哪个 Sleep 角色将获得资源。注意在该系统中没有公平机制，因此事实上可能只有一个 Sleep 角色会获得资源的使用权。

4.3 小结

本章描述了 PN 和 Rendezvous 两个域，这两个域中的角色都通过各自的线程来执行。只要 PN 模型没有包括 NondeterministicMerge 实例，那么它就是确定性的。会话域通常不是确定性的。当模型包含锁定角色（锁定时间不确定）时，为了不锁定整个模型，PN 就可以发效作用。对于资源管理问题，会话是特别有用的，尤其是对有限资源存在竞争并且对于模型中的并行行为需要进行时间同步的情况。

练习

这些练习的目的是为了加深了解进程网络计算模型的了解，熟悉它的编程与命令式模型的编程有何不同[⊖]。对于下面的所有练习，必须使用 PN 指示器和“简单的”角色来完成任务。比如下面的这些角色就足够了：

- Ramp 和 Const（在源库中）。
- Display 和 Discard（在 Sinks 中）。
- BooleanSwitch 和 BooleanSelect（在 FlowControl → BooleanFlowControl 中）。
- SampleDelay（在 FlowControl → SequenceControl 中）。
- Comparato、Equals、LogicalNot 或者 LgicGate（在 Logic 中）。

请随意使用“简单的”任何其他角色。同时，为测试用到的复合角色，请随意使用简单或非简单的任何其他角色，但是应使用简单的角色来实现复合角色。

1. SampleDelay 角色可产生初始令牌。在本练习中，将创建一个消费初始令牌的复合角色，因此它被认为是一种负延迟（negative delay）。

- (a) 创建一个 PN 模型包含一个复合角色，该角色包含一个输入端口和一个输出端口，其中输出序列和输入序列相同。也就是说，复合角色会抛弃第一个令牌，然后扮演一个类似恒等函数的角色。

⊖ 可能想用 -pn 选项来运行 Vergil，其给出一个 Ptolemy II 的子集。做这个练习只要要在命令行简单地输入“vergil -pn”。如果你正在从 Eclipse 运行 Ptolemy II，那么在 Java 视角的工具栏中，选择 Run Configurations。在 Arguments 选项卡中输入 -pn。

- (b) 增强该模型以便抛弃的初始令牌的数量能由复合角色的一个参数给出。提示：熟悉一个包括内置函数 `repeat` 的表达式语言[⊖]（见第13章）可能是很有用的。例如，

```
repeat(5, 1) = {1, 1, 1, 1, 1}
```

2. 下面的主要问题讨论依赖数据的数据流的操作。

- (a) 创建一个 PN 模型包含一个复合角色，该复合角色有一个输入端口和一个输出端口，其中输出序列和输入序列基本相同，但是如果一个序列中的令牌值都相同，那么输出将用值相等的一个令牌来替代输出该序列。也就是说，冗余的令牌将被删除。请按上述要求对模型使用合适的方法将结果演示。

- (b) (a) 模型能实现在有界缓冲区下永远运行吗？如果可以，请证明；如果这个要求是不可实现的，请解释原因。

3. 仅使用“简单的”PN 角色创建一个 `OrderedMerge` 角色的实例。（注意，请勿使用 Java 实现的 `OrderedMerge` 角色，（见第4章补充阅读：对 PN 模型有用的角色））。实现应该是有两个输入端口和一个输出端口的复合角色。当输入端口上存在任意两个数值递增的令牌序列时，角色应不丢失任何令牌，且将其合并到一个数值递增的序列中。如果两个序列所包含的令牌相同，则输出的顺序就无关紧要。

4. 图 4-9 中的模型将产生一个数字序列，该数字序列称为海明数字（Hamming number）。它们的形式是 $2^n 3^m 5^k$ ，并且它们在数值递增的次序（numerically increasing order）中产生，没有冗余。这个模型可以在 PN 演示中找到（标记为 `OrderedMerge`）。该模型可以在有界缓冲区中永久运行吗？请说明理由。

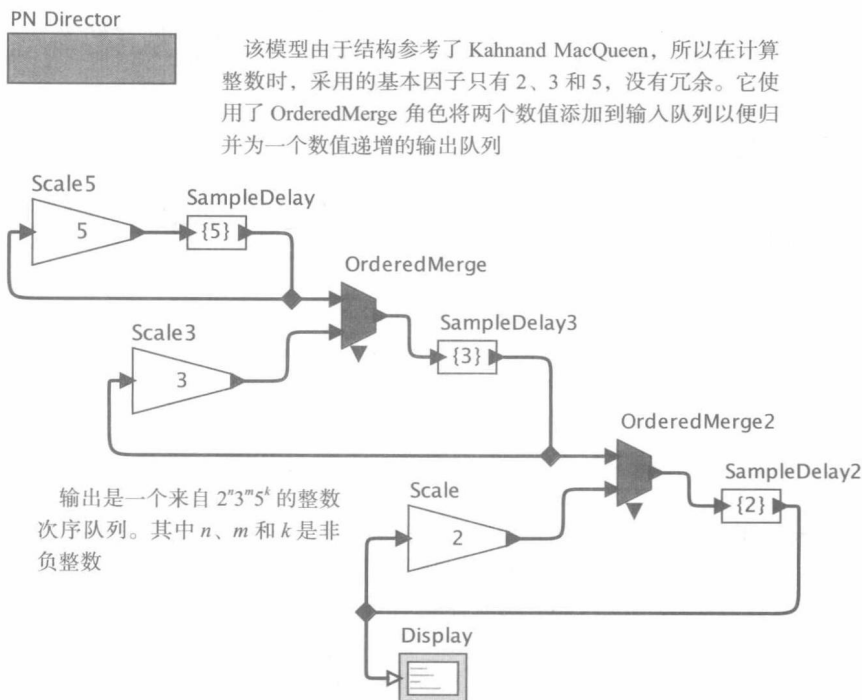


图 4-9 产生海明数序列的模型

针对这个问题，假设使用的数据类型是无限整数，而不是传统的 32 位整数。因为 32 位整数将

⊖ 注意，可以通过在 [File → New] 菜单中打开 `ExpressionEvaluator` 窗口，很容易地研究表达式语言。并且，单击在任何参数编辑窗口中的 `Hlep` 将提供表达式语言的文档。

很快溢出，超出可描述的范围而变成负数。

5. 嵌入式系统中的一个常见场景是多个传感器在不同的速率下提供数据，并且数据必须整合以便形成一个对物理世界一致的描述，通常情况下，该问题称为**传感器融合**（sensor fusion）。从带噪声传感器数据中形成一个一致的信号处理描述是非常复杂的，但是在本练习中，不将注意力集中在信号处理上，而是集中在并发和逻辑控制流上。在底层，传感器和嵌入式处理器通过硬件相连，这些连接使用的硬件会触发处理器中断，然后中断服务程序会读取传感器数据并将它存储到内存中的缓冲区。当提供数据的速率不同时会更加困难（尤其当它们的速率倍数关系都不是有理数，或者速率可能随时间变化，或者甚至可能非常不规律）

假设有两个传感器，传感器 A 和传感器 B，都对相同物理现象进行测量，这个物理现象恰好是对时间呈正弦函数，如下所示：

$$\forall t \in \mathbb{R}, x(t) = \sin(2\pi t/10)$$

假定时间 t 为一个较短时间，频率为 0.1Hz。进一步假定两个传感器通过不同的采样间隔对信号进行采样，以便得到以下度量：

$$\forall n \in \mathbb{Z}, x_A(n) = x(nT_A) = \sin(2\pi nT_A/10)$$

其中 T_A 为传感器 A 的采样间隔。同时对传感器 B 进行同样设置，其中样本周期为 T_B 。

图 4-10 中展示了使用 PN 指示器的传感器模型。可以在 Vergil 中通过调用 [Graph → Instantiate Entity] 菜单命令来创建一个传感器实例，并在如下框中填写：

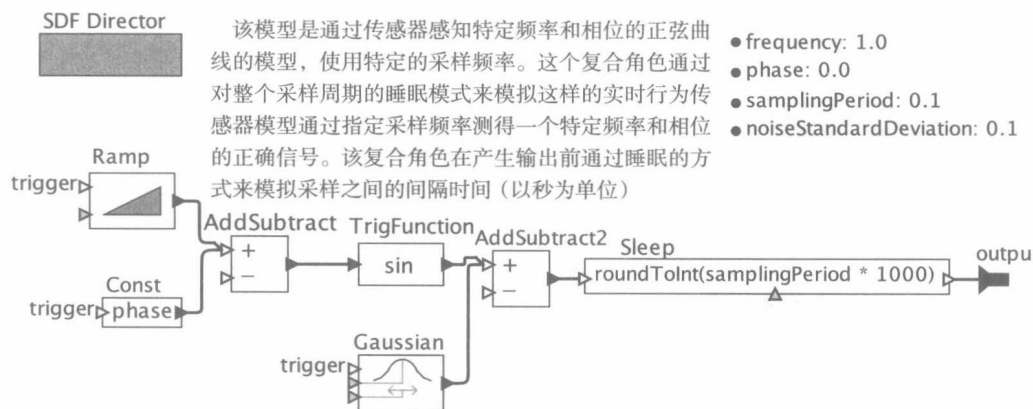


图 4-10 实时传感器模型

```
class: SensorModel
location (URL): http://embedded.eecs.berkeley.edu/
concurrency/models/SensorModel.xml
```

用 PN 指示器在 Ptolemy II 模型中创建两个传感器实例。

传感器有一些参数。frequency 需要设置为 0.1 以匹配上面的等式。将一个传感器实例的 samplingPeriod 设置为 0.5s，另一个设置为 0.75s。完成以下实验^①。

- 将每个传感器实例连接到各自的 SequencePlotter 实例。执行该模型。改变 SequencePlotter 参数以便 fillOnWrapup 为 false，并且可能需要将图的 X 轴的范围设置为“0.0, 50.0”（通过操作来更改）。描绘看到的情况。两个传感器能准确地反映正弦信号吗？为什么它们好像有不同的频率？
- 一个简单的传感器融合方法是简单地将传感器数据平均。构建一个模型，通过简单地求和并乘以 0.5 来平均从两个传感器得到数据。绘出结果信号。这个信号是对原始信号的一个更好的测

① 可以在第 2 章知识点：多速率数据流角色、信号处理角色和 StreamIt 中可以找到所描述的角色。

量值吗？请解释原因？这个模型可以在有界存储器中永远运行吗？请解释。

- (c) 对平均样本进行平均的传感器融合策略可以通过规范采样率来改进。给定的采样周期分别为 0.5 和 0.75，在 PN 中找到一种方法来实现它。如果样本周期不是这种简单的关系，比如，假设第一个传感器的周期是 0.500 001s，而不是 0.5s 该方法是否依然有效。
 - (d) 当传感器以不同速率收集数据且速率之间的关系并非简单关系时，一个可以证明有效的技术就是对数据创建时间戳（time stamp）并使用这些时间戳以改进度量。请建立一个实现它的模型，目标是生成一个能够以合理方法融合两个传感器数据的图。
6. 在该问题中，我们探讨如何使用 4.1.2 节的机制来确定性地停止一个 PN 模型的执行。具体地说，在每种情况下，我们认为一个 Source 角色提供一个可能是无穷的数据令牌序列给 Display 角色。我们希望给这个序列限定一个长度，并且希望保证 Display 角色可以显示序列的每一个元素。
- (a) 假设有一个 Source 角色，它有一个输出端口且没有参数，它的进程迭代一直产生输出。假设一个 Display 角色读取它的输出，这个 Display 角色有一个输入端口且没有输出端口。请找到一个使用 Stop 角色来确定性地停止该执行的方法，或者证明没有这样的方法。特别地，Source 角色应该产生一个指定数量的输出，并且每个输出应该在执行停止前由 Display 角色消耗和显示。
 - (b) Ptolemy II 中大多数的 Source 角色有一个 firingCountLimit 参数，该参数限制它们产生的输出数量。请演示在没有 Stop 角色帮助下，如何通过该参数可确定性地停止该执行。
 - (c) 在 Ptolemy II 中的许多 Source 角色拥有 trigger 输入端口。如果将这些输入连接，那么角色进程在产生每个输出之前将从输入读取一个值。请演示在有或者没有 Stop 角色的情况下如何使用这一机制来实现确定性地终止执行的目标，或者证明这是不可能实现的。此外，Source 产生一个预先指定数量的数据，并且 Display 应该消费和显示所有数据。可能使用 Switch、Select 或者任何其他合适的简单角色。一定要解释所用的每个角色，除非确信它确实是 Vergil 库所提供的角色。
7. 图 4-7 显示一个确定性地交错存取两个 Ramp 角色的输出。这个模型使用一个非确定性机制（Merge 角色的条件会话），然后使用多重会话和一个 Buffer 角色仔细地调节非确定性。最终的结果是确定性的。然而，同样的目标（用一个交替循环的方式确定性地交替两个流）可以用纯粹的确定性机制来完成。构建一个 Rendezvous 模型来完成这个功能。提示：通过使用 PN 或者 SDF 指示器而不是会话，模型的功能请保持不变。

同步响应模型

Stephen A. Edwards、Edward A. Lee、Stavros Tripakis、Paul Whitaker 为了纪念 Paul Caspi

同步是并发系统中的一个基本概念（见本章补充阅读：关于同步），**同步响应**（Synchronous-Reactive, SR）计算模型可以对涉及同步的系统建模，特别是那些拥有复杂控制逻辑的应用非常适合采用这种模型，在这类应用中，往往会有许多事件并行发生，然而**确定性**和准确控制却很重要。这类的应用必须包括能够保证安全性的嵌入式控制系统。SR 系统可以很好地调度并发行为、管理共享资源、检测和适应系统故障。虽然**数据流**模型可以很好地处理数据流，但是 SR 系统在处理偶发性数据时更具优势。产生这些数据的事件可能存在（present），也有可能缺失（absent），即使一个事件不发生，这本身也是有意义的**数据**（而不仅仅是数据传输的延时）。例如，检测一个事件是否为缺失（absent）就可能是**故障**管理系统中的重要组成部分。SR 对于协调有限状态机也是极其有效的计算模型。有限状态机可以用来表示并发执行角色的控制逻辑，这将在第 6 章和第 8 章中介绍。

Ptolemy II 的 SR 域已经被**同步语言**（synchronous language）体系（见本章补充阅读：同步响应语言）所影响，尤其是**数据流同步语言**（dataflow synchronous language），例如，Lustre（Halbwachs et al., 1991）和 Signal（Benveniste and Le Guernic, 1990）。SR 率先实现了 Edwards 和 Lee（2003b）提出的**同步框图**（synchronous block diagram）。这一计算模型与同步数字电路密切相关。尽管 SR 域更多地用于嵌入式软件的建模而不是电路建模，但本章将依旧用电路模型来举例。

SR 是一个**逻辑计时**（logically timed）系统。在这样的系统中，时间被处理成一系列的离散时间片，称为**响应**（reaction）或者**时钟节拍**（tick）。尽管这些时间片是有序的，但是在 SR 域中节拍之间并无离散时间系统中的“时延”概念，优先的没有 prioni 这种实时概念。因此，在 SR 域中提到的时间是**逻辑时间**（logical time）而不是离散时间。

SR 模型与**数据流**模型的相似点与不同点如下：

1）类似于**同构同步数据流模型**（Homogeneous SDF），SR 模型的迭代由模型中角色的迭代组成。模型的每次迭代都与逻辑时钟的时钟节拍同步。第 2 章中介绍的同步数据流模型大部分就是同步响应模型。例如，图 2-29 中的通道模型行为，它的所有变体在 SR 指示器中的行为都是一致的。

2）不同于数据流和**进程网络**，SR 在角色之间的通信没有缓冲区。在 SR 模型中，一个角色产生的输出由同一个时钟节拍内的目标角色来观察。与会话（rendezvous）不同在于，尽管会话也没有通信缓冲区，但 SR 是一个**确定性的**（determinate）模型。

3）不同于数据流模型，SR 模型在某个时钟节拍可能会出现输入或输出为缺失（absent）的情况。在数据流模型中，一个缺失（absent）输入仅仅表示这个数据在这个时钟节拍还没有到达。然而，在 SR 模型中，这种缺失（absent）信号有着更多的含义。它不是由计算或通信的时延或者调度故障造成的，相反某个时钟节拍上的缺失（absent）信号在 SR 模型中具有明确定义。因此，在 SR 模型中，角色可能会对缺失（absent）信号做出响应，而在数据流模

型中,角色不会响应这种缺失(absent)信号。

4) 正如下文即将介绍的,在SR模型中,角色在后点火(postfire)函数调用期间可能被多次点火。因为角色的一次迭代可能不仅仅由调用点火(fire)方法组成。对于那些简单模型,尤其是在那些无反馈的模型中,可能不引人注意,但有时候从中也可发现一些非常重要的细节。因此,本章将重点介绍这些看似简单的模型。

5.1 固定点语义

如图5-1a所示,这是一个拥有3个角色的模型,用 n 表示时钟节拍数。第一个本地时钟节拍定义为 $n=0$,第二个为 $n=1$,以此类推。每个角色在每个时钟节拍中都会执行一次从输入映射到输出的函数(该函数在每个时钟节拍都不同,因为它可能依赖于前面的输入)。例如,角色1在 $n=0$ 时执行函数 $f_1(0)$ 。也就是说在输入端口P1给定一个输入值 $s_1(0)$,它将在输出端口p5产生一个输出值 $s_2(0)=(f_1(0))(s_1(0))$ 。

输入信号在全球时钟内的任何时钟节拍都可能为absent。在这种情况下,“absent”可能被当作其他值处理。角色可以对absent输入做出响应,并且会声明一个absent信号输出或赋给输出,兼容输出端口数据类型的值。

因此,在每一个时钟节拍,每个角色产生一个值序列(或者是信号),角色1产生值 $s_2(0), s_2(1), \dots$,而角色2产生值为 $s_1(0), s_2(1), \dots$,角色3产生 $s_3(0), s_3(1), \dots$,这些值中的任何一个都可能是缺失(absent)信号的值。而SR指示器就要负责找出这些值(包括absent)。这就是执行该模型的意义。

如图5-1b~d所示,任何SR模型都可以重新排列成一个在第 n 个时钟节拍拥有函数 $f(n)$ 的单一角色模型。这个函数的域是由 $s(n)=(s_1(n), s_2(n), s_3(n))$ 的表示值(或absent)的一个元组(tuple)。所以这是上域(codomain)。因此,在第 n 个时钟节拍,指示器的任务就是找到 $s(n)$ 。

$$s(n)=(f(n))(s(n))$$

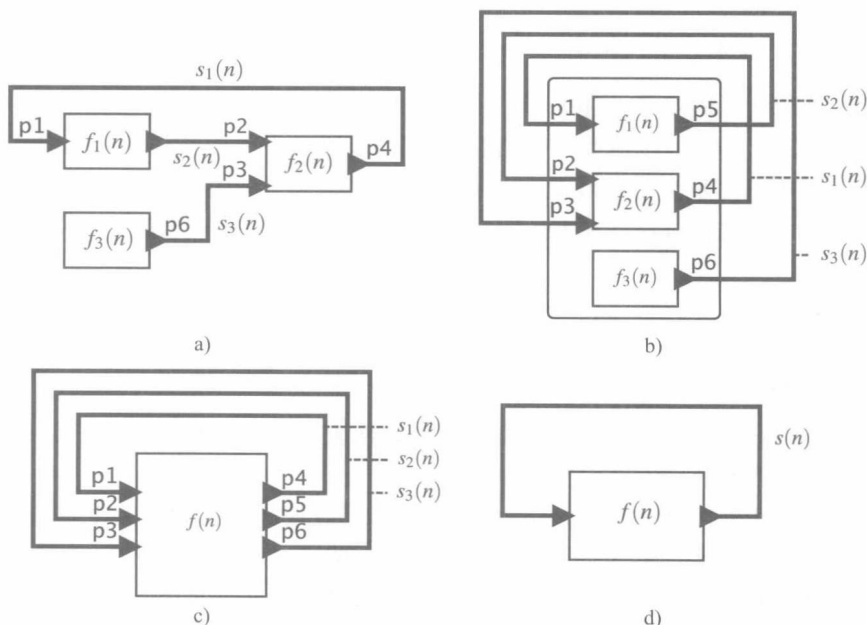


图5-1 SR模型的每个逻辑时钟节拍都可简化为固定点问题

在每个逻辑时钟节拍，SR 指示器寻找函数 $f(n)$ 的定点 $s(n)$ 。因此，SR 模型的微妙之处在于这样的固定点是否存在，以及这个固定点是否是唯一的。在一个构造好的 SR 模型中一定可以在有限的时钟节拍内产生一个唯一的固定点。

逻辑上，SR 模型从概念上可以看成在每一个时钟节拍上所有角色的同时且瞬时 (simultaneous and instantaneous) 响应。“同时”强调了所有的角色都在同一时刻做出响应。“瞬时”则意味着每个角色的输出与输入同一时刻产生。输入和输出是定点解的一部分。假设有这样一个理想的模型，在这个模型中，角色的执行时间可忽略不计，这个理想模型就称为同步假设 (synchrony hypothesis)。但是，这个模型并非想象的那么简单，因为该模型中还存在反馈，那么一个角色需要响应的输入是其自身函数的输出。这可能会陷入因果 (causality) 关系问题。因为只有知道了输出才可能知道输入，只有知道了输入才可能知道输出。而这一矛盾就是 SR 模型中的一大难题。

图 5-2 中的模型是一个没有反馈的简单 SR 模型。该模型甚至可以化简为一个定点 (fixed-point) 问题，但这样会显得过于简单。从 SR 指示器的角度看函数 $f_1(n)$ 只需在每个时钟节拍计算一次就会立刻找到在时钟节拍 n 上的固定点，而不需要再计算 $f_2(n)$ 。但是 SR 指示器无论如何都会点火且后点火 (postfires) 角色 2，因为它可能有边界效应 (例如，刷新显示)。但是角色 2 在寻找固定点上没有起到作用。

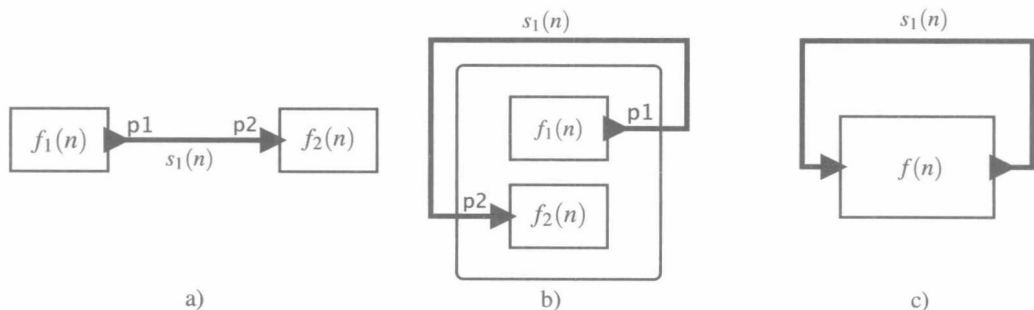


图 5-2 甚至不带反馈的 SR 模型的每个逻辑时钟节拍也可简化为固定点问题

一旦指示器找到了固定点，在为下一个时钟节拍做准备时，它将把模型中每个角色的函数更新为 $f(n+1)$ ，一般发生在模型执行的后点火 (postfire) 阶段。因此，在找到固定点前，模型每次迭代由角色的多次点火组成，之后调用后点火函数，由固定点提供的输入响应来更新角色状态。迭代的执行过程将在 5.3 节中详细介绍。在此之前，先来看一些例子。

5.2 SR 实例

5.2.1 非循环模型

没有反馈的 SR 模型与没有反馈的同构同步数据流模型非常相似，但是 SR 模型允许信号为 absent，这使得控制角色执行非常方便。

例 5.1 回到图 3-10 中与 if-then-else 编程结构等效的模型，它使用动态数据流有条件地将令牌路由到计算端口。相似的效果也可以在 SR 模型中用 When 和 Default 实现 (见第 5 章补充阅读：特定域的 SR 角色)，如图 5-3 所示。该模型由 Ramp 角色以两种方法中的一种 (相对简单的那种) 来产生输入数据。数据流经过上方路径，乘以 -1 。经过下方路径，则它

乘以 1。该模式可能用来对系统中的间歇性故障建模。

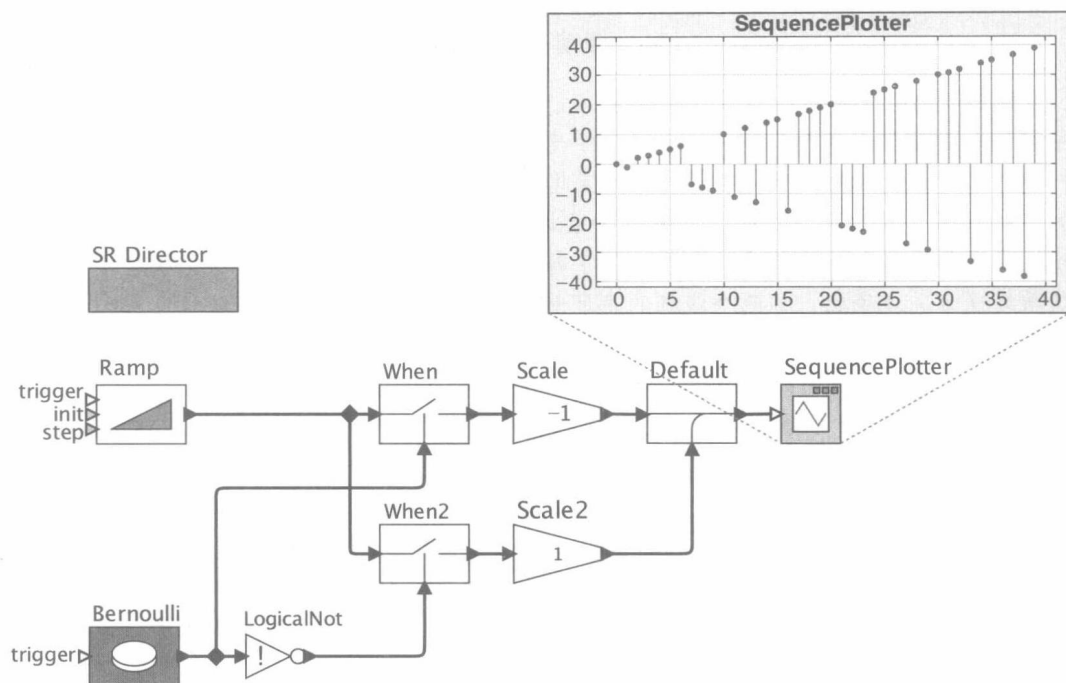


图 5-3 通过 SR 实现条件执行的模型

Bernoulli 角色产生一个随机的布尔值，用来控制两个 When 角色。当布尔值为 true 时，Ramp 产生的输出将上方的 When 角色发送到它的输出端口，反之，Ramp 产生的输出将下方的 When 角色发送到它的输出端口。当 When 的输出信号为 absent 时，后面的 Scale 角色的输出信号也会是 absent。因此，Default 角色在每个时钟节拍只有一个输入信号。当 Default 角色收到这个输入信号后，它将这个输入发送到它的输出。最后，由 SequencePlotter 绘制出结果。

如图 3-13 所示的数据流模型中，有可能会出现接线错误，导致无界缓冲区；而在 SR 模型中，执行总是有界的。两个角色之间的每个连接都在一个时钟节拍内最多存储一个令牌。因此，对于内存使用，SR 模型总是有界执行（除非某个角色的内部是无界的）。

5.2.2 反馈

如图 5-1 所示，许多 SR 模型都包含反馈（在图中表示为有向环）。在这样的反馈系统中，因果（causality）关系是需要关注的问题。试想图 5-1a 中的角色 1 与角色 2 之间的关系。在第 n 个逻辑时钟节拍，为了计算 $f_1(n)$ 的值，就必须知道 $s_1(n)$ 。而要知道 $s_1(n)$ ，又必须得到 $f_2(n)$ 。要知道 $f_2(n)$ 又必须先知道 $s_2(n)$ ，而知道 $s_2(n)$ 的前提是计算出了 $f_1(n)$ 的值。这样就陷入了因果循环（causality loop）。

必须通过非严格角色（non-strict actor）来打破这样的因果循环。如果一个角色只有得到了它所需要的所有输入才能产生输出，那么这样的角色就是严格的（strict）。反之，如果一个角色可以在不知道它所有所需要的输入的情况下也能产生输出，那么这样的角色就是非严格的。最简单的非严格角色是 NonStrictDelay（见第 5 章补充阅读：特定域的 SR 角色）。在

后续例子中，可以用它来打破因果循环。

补充阅读：关于同步

同步 (synchronous) 一般有两个定义：1) 发生或存在于同一时刻；2) 以相同的速率移动或者操作。在工程学和计算机科学中，大部分情况下同步的含义与这两个定义是一致的，但是，也有一些同步的定义与前面两个定义不一致。当提到使用线程和进程的并发软件时，同步通信指的是一种称为**会话** (rendezvous) 的通信，在这样的通信方式中，信息的发送方必须等待接收方做好了接收信息的准备才能发送信息，接收方也必须等待发送方。理论上，从两个线程的角度看，通信是同时发生在两个线程之间的。这与定义 1) 一致。然而，在 Java 中，关键字 `synchronized` 被定义为阻塞那些不允许同时执行的代码，这一定义与以上两个定义都是不同的。

本章使用同步为第三种定义，该定义是**同步语言** (synchronous language) 的基础 (见第 5 章补充阅读：同步响应语言)。这些语言包含了两个关键的概念：第一，程序中组件的输出与它们的输入在概念上是同时出现的 (这称为**同步假设** (synchrony hypothesis))；第二，程序中组件的执行在概念上是**同时且瞬时的** (simultaneously and instantaneously)。虽然，这在现实生活中是不可能实现的，但正确执行必须看起来完成了同步。对同步的第三种解释与前面的两个定义是一致的，因为程序中所有组件的执行都需要在同一时刻，且以相同的速度进行操作。

在电路设计中，同步指的是一个时钟信号在整个电路中使得电路单元“**锁存器**” (latches) 在时钟的上升沿或下降沿瞬间记录锁存器的输入状态的这样一种形式，当然上下时钟沿的时间必须足够使锁存器之间的门电路稳定下来。概念上，这个模型与同步语言中的模型非常相似。假设锁存器之间的门电路与同步假设一样没有延时，并且全局时钟分布使得这些门电路的执行是同时且瞬时的。因此在数字电路建模中 SR 模型非常有效。

在能源系统工程学中，同步是指电波有相同的频率与相位。在信号处理中，同步意味着信号有相同的采样率，或者采样率是另一个信号采样率的固定倍数。在 3.1 节中介绍的**同步数据流**中的同步是建立在与定义 2) 一致的基础上。

补充阅读：同步响应语言

同步响应计算模型有很长的历史，至少可以追溯到开发编程语言的 20 世纪 80 年代中期。术语“响应”来自于**转换系统** (transformational system) 和响应系统之间有所区别的计算机系统。转换系统接受输入数据、执行计算并产生输出数据。而响应系统与环境保持不断的对话 (Harel and Pnueli, 1985)。Manna and Pnueli (1992) 指出：

“响应程序的作用……不是得到一个最终结果，而是保持与环境不间断的对话。”

转换系统与响应系统的区别导致一系列创新性编程语言的发展。**同步语言** (synchronous language) (Benveniste and Berry, 1991) 采用了一种特殊的方法来设计响应系统，其中程序的各个部分在全局时钟的每个节拍**同时且瞬时地**进行响应。这些语言

还有 Lustre (Halbwachs et al., 1991)、Esterel (Berry and Gonthier, 1992) 和 Signal (Le Guernic et al., 1991)。statechart (Harel, 1987) 及其在 Statement (Harel et al., 1990) 中的实现都有很强的同步意味。

SCADE (Safety Critical Application Development Environment, 高安全性应用开发环境) (Berry, 2003) 是 Esterel 技术公司的商业化产品, 它是在 Lustre 的基础上, 借用 Esterel 概念, 并提供一个图形语法建立起来的, 在其中描绘了与第 6 章中模型相似的状态机模型, 并将角色模型同步的组合在图中。同步语言引人注目的原因是其具有强大的形式化特性, 可以进行有效的形式化分析和技术验证。出于这个原因, SCADE 模型被 Airbus 公司用于商用飞机的高安全飞行控制软件系统的设计。

在 Ptolemy/II 中, SR 是协调语言 (coordination language) 形式而不是程序语言。(见 ForSyDe (Sander and Jantsch, 2004), 它还使用协调语言中的同步。这使得系统中的“原语”成为复杂的组件, 而不是内置语言原语。这种方法促进了 MoC 的异构组合, 因为复杂的组件本身还可以由其他 MoC 的组件组合而成。

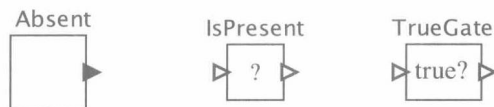
补充阅读：特定域的 SR 角色

下面 DomainSpecific \rightarrow SynchronousReactive 中的 SR 角色是受到同步语言 Lustre 和 Signal 中相应的操作符的启发后创建的。



- **Current**: 输出最近收到的非 absent 信号。如果没有收到输入信号, 那么输出信号为 absent。
- **Default**: 用一个优先级合并两个信号。如果首选输入 (左边的输入) 是 present, 那么输出等于这个输入。如果首选输入是 absent, 那么输出等于备用输入 (底部的输入, 不管这个输入是否为 absent)。
- **NonStrictDelay**: 提供一个时钟节拍时延。每次点火时, 不管在前一个时钟节拍输入端口的输入值是什么, 在输出端口都会产生一个时延。如果输入在前一个时钟节拍为 absent, 那么输出为 absent。在第一个时钟节拍, 值由 initialValue 参数给出。如果没有给 initialValue 赋值, 那么第一个输出为 absent。
- **Pre**: 输出前一个时钟节拍接收的 (非 absent) 输入。当输入是 absent 时, 输出也为 absent。第一次的输入为 present, 则输出由角色的 initialValue 参数给出 (这个参数的默认值为 absent)。与 NonStrictDelay 相反, Pre 是严格的, 这意味着在确定输出时, 输入必须是已知的。因此, 它不会打破因果循环。如果要打破因果循环, 就要使用 NonStrictDelay。
- **When**: 根据一个信号来过滤另一个信号。如果控制输入 (底部的输入) 是 present 且为 true, 那么将数据输入 (左边的输入) 复制给输出。无论控制输入为 absent, 或者为 false 或 true, 如果数据输入为 absent, 那么输出为 absent。

Ptolemy II 库也提供多个角色来操作 absent 值：



- Absent: 输出永远是 absent。
- IsPresent: 如果输入是 present, 则输出为 true; 否则, 输出为 false。
- TrueGate: 如果输入是 present 且为 true, 那么输出为 true; 否则, 输出为 absent。

例 5.2 如图 5-4 所示为一个简单的数字电路模型。它是一个 2 位的模 4 计数器模型, 它产生整数序列 0, 1, 2, 3, 0, 1... 反馈回路使用 NonStrictDelay 角色, 每一个 NonStrictDelay 角色对一个锁存器 (锁存器是用于值存储一段时间的电子器件) 建模。这个模型还包括对逻辑门建模的, 两个角色, 这两个角色是 LogicalNot 和 LogicGate (见第 3 章补充阅读: 逻辑角色)。

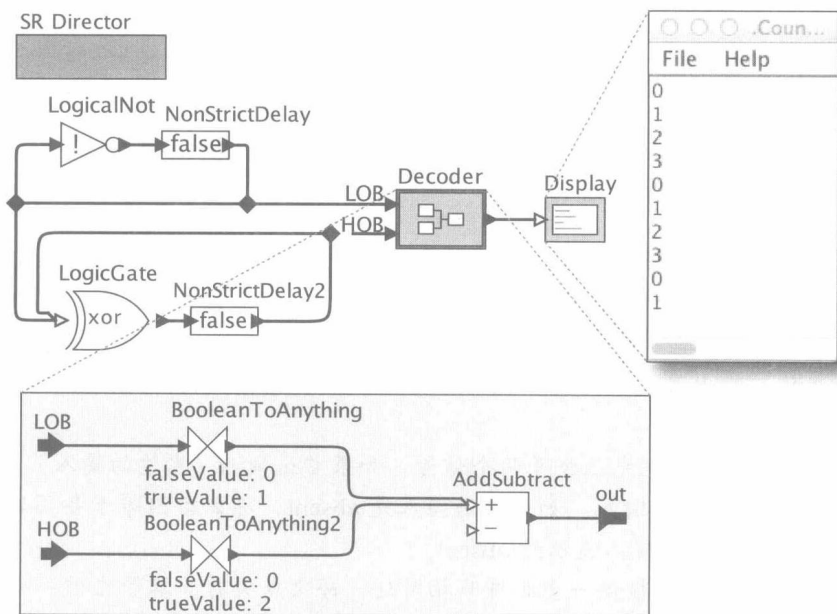


图 5-4 SR 中的一个 2 位计数器模型。顶层模型包含一个可将布尔数据转换为整数的解码器复合角色

回路的上方, 包含 LogicalNot, 用来对计数器的低位 (LOB) 建模。它的起始值为 false, NonStrictDelay 的初始输出, 在后面的时钟节拍中它的值将是 true 和 false, 交替出现。

在下方的回路中, 包含 LogicGate, 用来对进位电路建模, 它实现计数器的高位 (HOB)。它的初始输入也是 false, 并且当 LOB 为 true 时, 它从 true 切换到 false, 或者从 false 切换到 true。

解码器 (Decoder) 是复合角色, 它负责产生一个可读性更强的显示。通过给 LOB 赋值, 它将两个布尔值转换为 0 ~ 3 的数值。它包含了两个 BooleanToAnything 角色, 这个角色将布尔值转换为 LOB 和 HOB 的值, 然后将它们相加。

图 5-4 中的角色 NonStrictDelay 是非严格的。它可以在不知道输入的情况下产生输出。在第一个时钟节拍, 输出值由角色的 initialValue 参数给出, 接下来将输出来自上一个时钟

节拍的输入，而这个输入在寻找固定点时可以确定。因此，这些角色打破了潜在的因果循环风险。

注意，如果用 Pre 角色来代替 NonStrictDelay 角色将没有效果，（见第 5 章补充阅读：特定域的 SR 角色）。Pre 角色是严格的，因为它必须知道它的输入是 present 还是 absent 才能决定输出是什么^①。

图 5-4 中的模型非常简单，并没有显示出 SR 模型的强大。事实上，它如果使用 SampleDelay 角色代替 NonStrictDelay 角色，这个模型还可以与 SDF 指示器一起执行。

SR 中的每一个有向循环需要至少包括一个非严格角色。但是，NonStrictDelay^②并不是唯一的非严格角色。例如，NonStrictLogicGate 也是一个非严格角色，它可以实现非严格的逻辑 AND（与）运算，所以也叫作并行 AND。有两个输入的非严格逻辑 AND 运算的真值表如下所示（事实上，这个角色可以接受任意数量的输入）。

表中的符号 \perp 表示未知（unknown）。通过上面的真值表可知，当输入是 false 时，不管它的另一个输入是什么（包括输入为未知（unknown）的情况），输出都是 false。

输入	\perp	true	false
\perp	\perp	\perp	false
true	\perp	true	false
false	false	false	false

例 5.3 尽管图 5-5 中的模型有反馈回路，但是其语意是确定性的。NonStrictLogicGate 角色实现了 AND 逻辑功能。因为它的输入总是 false，所以每个时钟节拍它的输出都是 false。

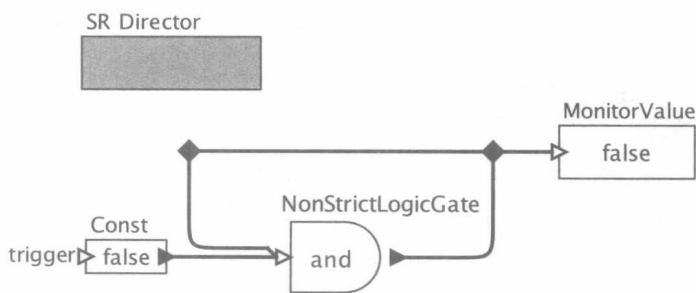


图 5-5 一个使用非严格逻辑 AND 的确定性模型

下面也是一个没有 NonStrictDelay 角色的带循环的实例。

例 5.4 图 5-6 是一个用 SR 实现的令牌环媒体访问控制协议（MAC），这个协议由 Edwards 和 Lee（2003b）提出。顶层模型中有 3 个 Arbiter 类的实例连接成一个循环。还有一个 ComposeDisplay 复合角色，它用来将执行结果转化成可读的形式显示在底部。

这个系统的目标是通过沿着环传送令牌实现共享资源访问权限的公平分配。在每个逻辑

① Lustre 同步语言（Halbwachs et al., 1991）能够通过使用时钟积分（clock calculus）将 Pre 变成非严格角色。时钟积分可以分析模型来确定在哪个时钟节拍输入是 present。因此，在 Lustre 中在输入为 absent 时 Pre 不执行。因此，当它执行时，尽管它不知道输入的值，它也知道输入是 present，这样它就可以产生输出。而 Ptolemy II 中的 SR 监视器没有实现时钟积分，而是采用 Esterel（Berry and Gonthier, 1992）中更为简单的时钟机制。

② 在执行的初始阶段 SampleDelay 产生一个初始输出。在数据流域中，这些初始输入都放在缓冲区中，在整个执行期间都可用。而在 SR 中，没有数据缓冲区，在初始化期间产生的输入都丢失。因此，SampleDelay 不能在 SR 中使用。

时钟节拍，如果持有令牌者请求访问，那么仲裁器给那些持有令牌者访问权限。如果持有令牌者没有请求访问，那么就将访问权限给第一个发出访问请求的请求者。在图中，3个请求者总是在不停地请求访问，而在底部的图中可以看出，这3个请求者轮流获得了访问权限。在这个模型中，InstanceOfArbiter1 最开始持有令牌（见实例的参数）。

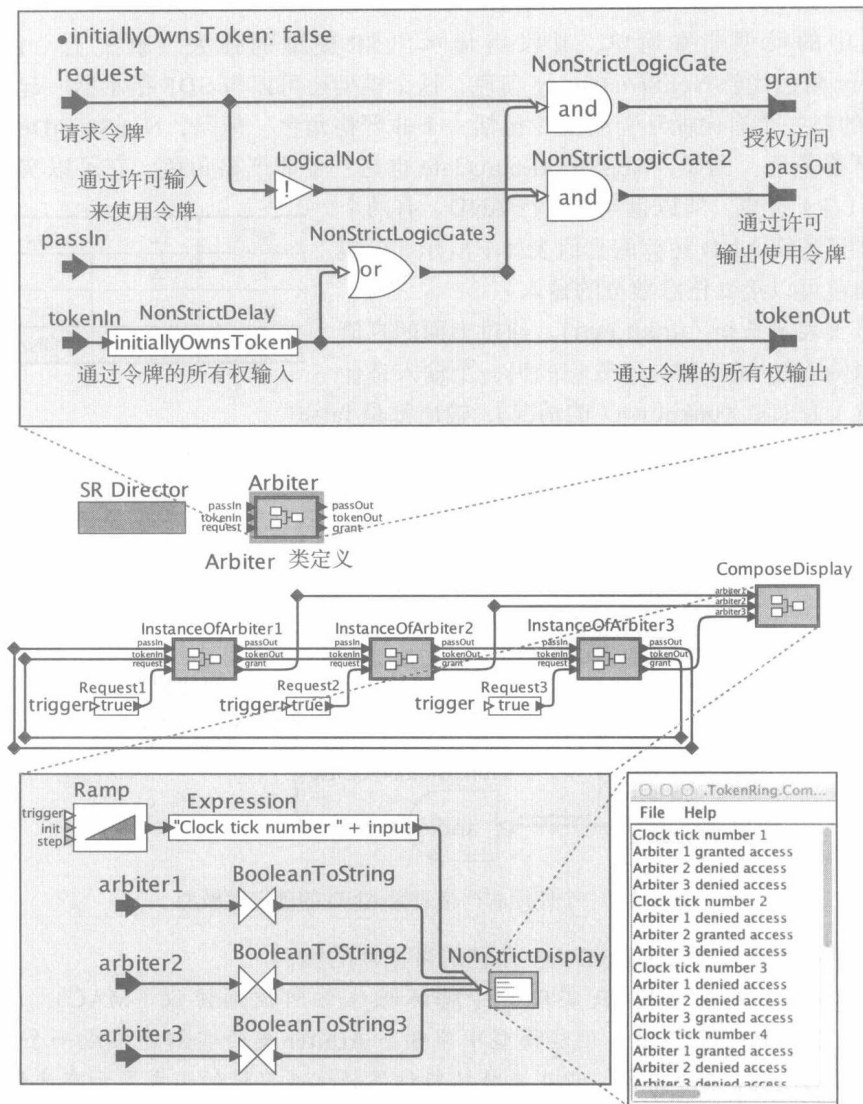


图 5-6 采用 SR 实现的令牌环介质访问协议（由 Edwards and Lee (2003b) 提出）

这 3 个仲裁器是面向角色的类（actor-oriented class）的实例，显示在图的上方。该类包含 3 个输入和 3 个输出。该类有一个 NonStrictDelay 实例，一旦这个仲裁器获得令牌，NonStrictDelay 就输出 true。这 3 个仲裁器中只有 NonStrictDelay 角色初始化为 true，每个时钟节拍仲裁器都将令牌传送给下一个仲裁器，这就形成了一个包含 3 个 NonStrictDelay 角色的循环。

但是，这里还有另一个没有 NonStrictDelay 角色的循环。这个循环通过每个仲裁器角色的 request 输入和 grant 输出。另外，这个循环还包含 3 个 NonStrictLogicGate 实例，用于实

现并行与运算。这个逻辑门负责将访问权限赋予那些满足以下条件的请求者。第一，发出了访问请求；第二，持有令牌或者它的 `passIn` 为 `true`（这意味着上游的仲裁器持有令牌但是没有发出请求）。这个循环是无法一眼看出来的，而且逻辑门的每次可以在不知道所有输入的情况下得到输出，所以这个模型不会遇到因果循环的情况。

另一种非严格角色是 `Multiplexor` 或者 `BooleanMultiplexor`（见第3章补充阅读：令牌流控制角色）。这些角色必须知道控制输入（图标的下方），然后确定将哪个输入数据传送到输出端口。这个角色只有知道了控制输入后才有可能产生输出。

例 5.5 图 5-7 是一个非常有趣的例子。这个模型通过由 `Bernoulli` 角色产生的类似掷硬币的方法来决定是计算 $\sin(\exp(x))$ 还是计算 $\exp(\sin(x))$ 。Malik (1994) 说这像是循环组合电路 (cyclic combinational circuit)，因为尽管该模型中存在着反馈，但是这个系统中没有状态保存。输出（每个绘制的点）只依赖于当前的输入（来自 `Ramp` 和 `Bernoulli` 的数据）。如果一个电路的输出只依赖于当前输入，与以前的输入没有关系，则这样的电路称为组合 (combinational) 电路。大部分带有反馈的电路不是组合电路。输出不仅依赖于当前的输入，也依赖于当前的状态，而当前的状态是随时间变化的。

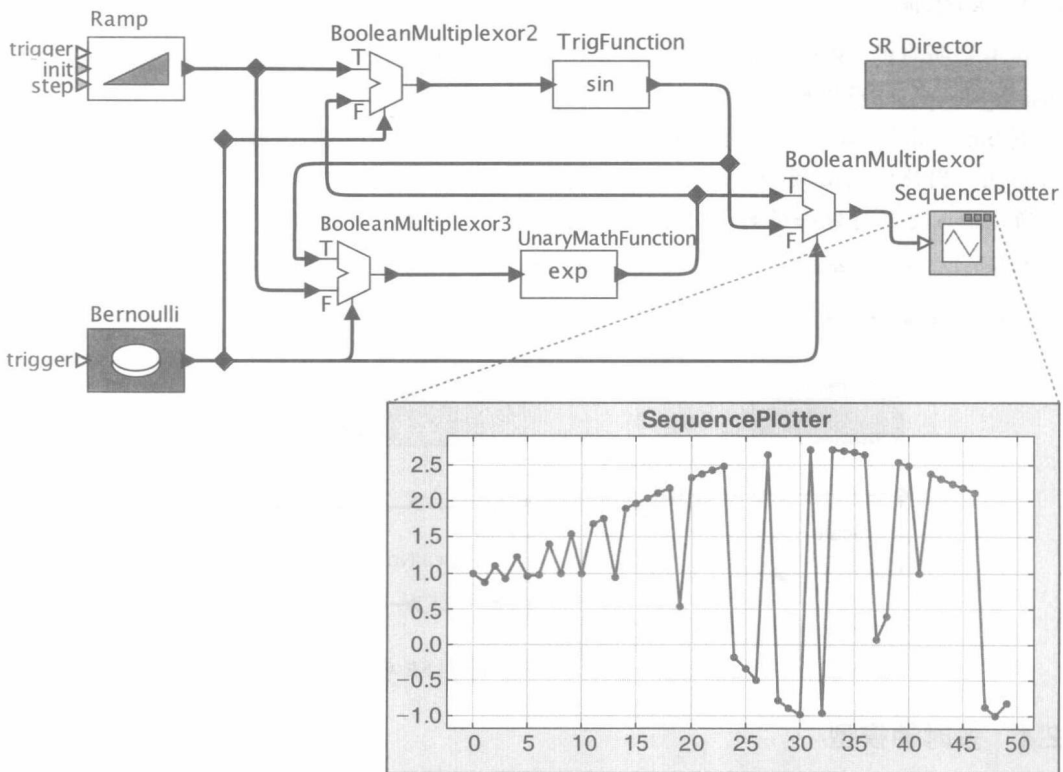


图 5-7 在 SR 中实现的循环组合电路模型（出自 Malik (1994)）

在这种情况下，反应用于避免两个负责计算的角色（`TrigFunction` 和 `UnaryMathFunction`，见第2章补充阅读：Math 库）有两个副本。图 5-8 是用 `TrigFunction` 和 `UnaryMathFunction` 实现的等效模型。如果用图 5-7 中每个角色都有用一个单独循环的方式来搭建的电路，那么比图 5-8 中的方式更节约成本。

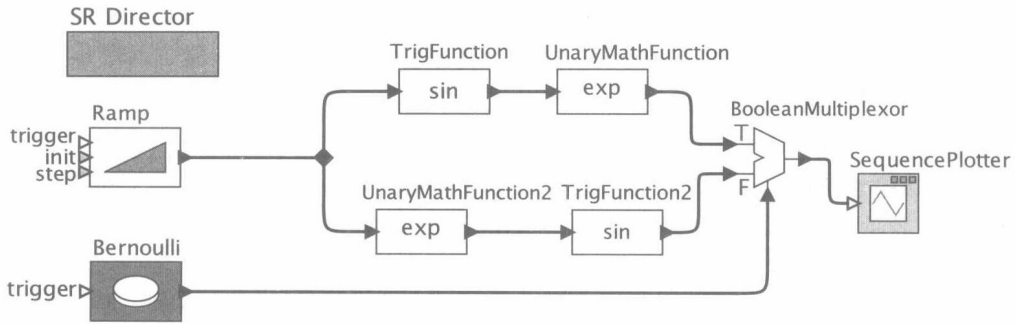


图 5-8 仅用两对数学角色来实现的图 5-7 的无循环等效模型

图 5-7 中的模型有 3 个 BooleanMultiplexor 角色，根据控制输入（在该角色的下部）设置为 true 还是 false，这些角色给输出端口发送“T”或“F”输入值。在每个时钟节拍，左边两个 BooleanMultiplexor 中的一个一定可以产生一个输出（一旦得到来自 Ramp 的输入）。因此，这个产生了输出的 BooleanMultiplexor 就打破了因果循环，并且可以找到定点。

5.2.3 因果循环

并不是所有的 SR 模型都是可执行的，尤其是下面例子中的那些可能构建出现因果循环（causality loop）的反馈模型。

例 5.6 图 5-9 显示了带无解的循环依赖关系回路的两个例子。Scale 和 LogicalNot 都是严格角色，因此它们必须知道输入才能决定输出。但是在这两个模型中，输入就是它们的输出，所以输入是不可能知道的。SR 指示器将拒绝这些模型，并抛出如下异常：

```
IllegalActionException: Unknown inputs remain. Possible
causality loop:
in Display.input
```

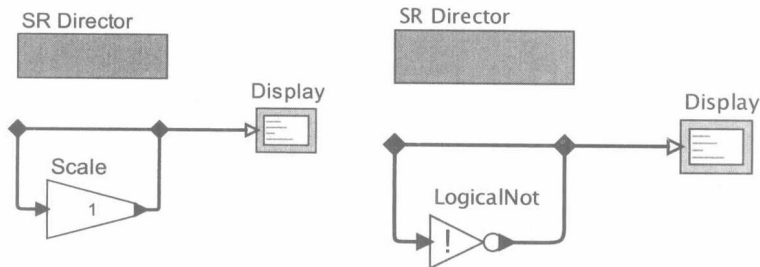


图 5-9 两个带无效回路的 SR 模型

5.2.4 多时钟模型

SR 域中只有一个逻辑时钟，是一个单独的全局时钟。SR 指示器控制的每个角色都在这个时钟的每个时钟节拍点火。但是，如果希望其中的某些角色被点火的频率不一样应该怎么办？幸运的是，Ptolemy II 中的分层（hierarchy）机制能很容易地构建一个以不同速率处理的多时钟模型，EnabledComposite 角色对构建这样的多时钟（multiclock）模型特别有效。

例 5.7 图 5-10 的条件计数（guarded count）模型，它从一个初始值开始递减计数到 0，然后将计数器的值更新为另一个新的值。顶层模型有两个复合角色和两个 Display 角色。

CountDown 复合角色使用 SR 原子角色 (primitive actor) 来实现符合如下描述的递减计数功能: 1) 无论什么时候, 只要它的 start 输入端口接收到一个值为 n 的 non-absent 输入, 它就 (重新) 开始从 n 向递减数; 2) 它的 count 输出端口输出 $n, n-1, \dots, 0$ 这样的值序列。当计数值等于 0 时, ready 端口输出 true, 表明这个角色已经准备好重新递减计数了。

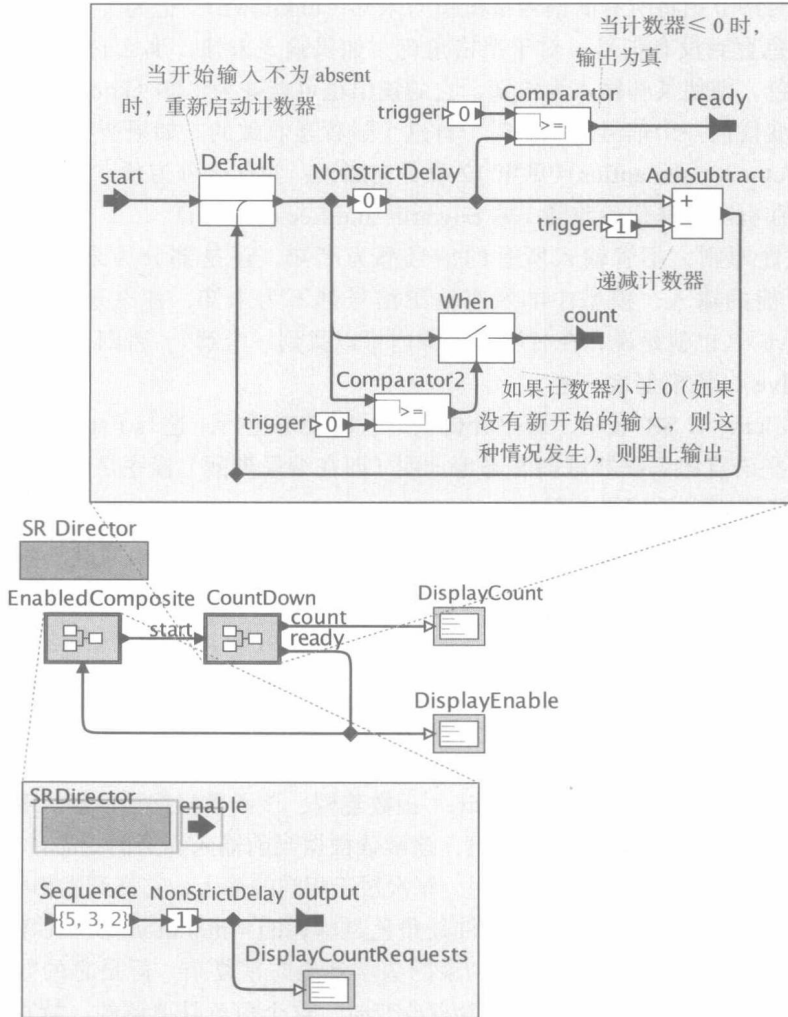


图 5-10 SR 中的多时钟模型

ready 信号控制 EnabledComposite 角色的点火。在这个复合角色中, 只有在 enable 输入端口接收到 true 信号时 (这个输入端口在角色的下方) 才响应。需要注意的是, ready 信号的初始值为 true, 因为 CountDown 中使用 NonStrictDelay 角色。

EnabledComposite 中的 SR 指示器的时钟比顶层 SR 指示器的时钟慢。事实上, 这两个速率之间的关系可以根据 Sequence 角色提供的数据动态决定。

5.3 寻找定点

有效地执行无循环模型 (如图 5-8 所示) 或每个循环可以被 NonStrictDelay 角色 “打破” 的循环模型 (如图 5-4 所示) 是很容易的。模型的角色可以根据它们的依赖关系进行排序

(例如, 使用拓扑排序算法), 然后按照这个顺序点火。在这种情况下, 每个角色只需要在逻辑时钟的每个时钟节拍被点火一次。

然而, 这种策略对类似图 5-6 和图 5-7 那样的模型不起作用, 因为角色的点火顺序依赖于角色计算出来的数据。幸运的是, 这类模型有一种简单的执行策略, 该策略的关键是在逻辑时钟的每个时钟节拍将所有的信号都赋值为未知 (unknown), 记为 \perp 。然后, 指示器以任意顺序评估角色直到没有信号。对于严格角色, 如果输入未知, 那么它的输出也是未知的。对于非严格角色, 即使某些输入为未知, 它的输出也可能变为已知 (known)。当所有角色的点火都不会改变任何一个信号的状态时, 称这个过程是收敛的。如果所有的角色都遵循严格角色语义 (strict actor semantics) (见第 12 章补充阅读: 为什么分为预点火、点火和后点火), 那么这个过程在有限次迭代后收敛 (见 Edwards and Lee (2003b))。

根据收敛性原则, 不管输入所有的信号都为已知, 还是部分为未知。只要在每次迭代时对所有可能的输入, 模型中的所有输出信号都不为未知, 那么这个模型就是**结构性的** (constructive) (也就是说, 在有限的步骤内可以找到一个解)。否则, 模型是**非结构性的** (non-constructive), 该模型被拒绝。

注意在 Ptolemy II SR 域中, 因果分析在运行时动态进行。这与 Esterel 这样的语言是不同的, 使用这些语言的编译器试图静态地证明 (即在编译期间) 程序是结构性的 (见第 5 章补充阅读: 同步语言中的因果关系)。

只有角色遵循严格角色语义的情况下 SR 才能正确地工作。为了更好地理解这一点, 可以将角色看作一个状态机。令 \vec{x} 、 \vec{y} 和 \vec{s} 分别表示输入矢量、输出矢量和状态矢量。状态机可以描述如下:

$$\vec{y}(n) = f(\vec{x}(n), \vec{s}(n)) \quad (5-1)$$

$$\vec{s}(n+1) = g(\vec{x}(n), \vec{s}(n)) \quad (5-2)$$

n 为逻辑时钟的时钟节拍号, f 对点火 (fire) 函数建模, 该函数根据当前输入和当前状态计算输出, 而 g 对后点火 (postfire) 函数建模, 该函数根据当前输入和当前状态计算下一个状态。这里的关键是, 点火函数可以被重复调用, 每次对于相同的输入, 它都产生相同的输出。

模型能够正确执行的另一个附加条件是角色是**单调的** (monotonic) (见第 5 章补充阅读 CPO、连续函数和固定点)。尽管这个约束的数学理论非常复杂, 但是它的数学表达式却非常简单。当给出更多的输入信息而输出没有改变时, 这个角色是单调的。特别是, 如果用未知输入调用点火函数, 那么当角色是非严格角色时, 它有可能产生输出。假设在这种情况下, 给出另一些更多的输入信息 (很少的输入是未知) 没有使得它产生与更少输入信息不同的输出, 那么这个角色是单调的。

大部分 Ptolemy II 角色都遵循严格角色语义且是单调的, 因此它们可以在 SR 中使用。

补充阅读: 同步语言中的因果关系

如何解决循环依赖关系的问题, 即因果问题 (causality problem), 是同步语言中的一个重大挑战。本章简单总结了几种方法, 相关研究文献和更多的细节, 读者请自行参阅 Caspi 等的文章。

最直接的解决方法是禁止所有的循环依赖关系。这种方法被 Lustre 语言采用，它要求每个数据流回路必须包含至少一个 pre 操作。同样的效果在 Ptolemy II SR 指示器中通过要求每个回路必须包含至少一个 NonStrictDelay 角色中实现，这个角色打破了瞬时循环依赖关系。Lustre 编译器静态地进行这一条件的检查，违背了这一条件的程序将无法被编译通过。该策略也被 SCADE 采用。

另一种方法是允许编译器接受条件更为宽松的结构性程序，Ptolemy II SR 域采用的就是这种方法。该方法首先由 Berry (1999) 为 Esterel 提出。Esterel 与 SR 的关键不同点在于，SR 中的定点在运行时计算（逻辑时钟的每个时钟节拍），而 Esterel 编译器的图在编译时证明程序是结构性的。后者通常更困难，因为程序的输入在编译时通常是未知的。另一方面，静态证明程序是结构性的有两个优点：第一，这对于安全至上的系统十分重要，因为这样可以避免运行时出现异常；第二，实现后的产品可以最小化运行时寻找固定点时的迭代开销。

还有一种方法是只接受确定性程序。或者，反之，如果一个程序可以被一个约束集表示，而这个约束集又没有唯一的解，那么该程序被拒绝。这种方式被 Signal (Benveniste and Le Guernic, 1990) 以及 Argos (Maraninchi and Rémond, 2001) 采用。这种方法的一个缺点就是某些可疑的程序也能够通过编译。例如，以下等式表示的系统程序：

$$Y = X \wedge \neg Y$$

尽管这个系统在古典的二值逻辑上有唯一的解，即 $X = Y = \text{false}$ ，但是它相应的实现是否有意义是不确定的。事实上，这样的简单组合 (combinational) 电路是不稳定的，其是一个振荡电路。

5.4 定点逻辑

回忆图 5-9 中的两个模型，这两个模型都有因果循环。然而，这两个模型也有不同。他们的不同表现在同步模型中的确定性语义和结构性 (constructive) 语义之间的不同。结构性语义建立在直觉逻辑 (intuitionistic logic) 思想上的模型是确定性的，但结构性语义会拒绝这个基于经典逻辑 (classic logic) 构建的模型，即使这些模型被广义确定性语义所接受。

可以将图 5-9 中左边的模型解释为如下等式。假设 Scale 角色输入与输出之间的关系为 x ，则

$$x = 1 \cdot x$$

如果以经典逻辑来理解这个等式，那么这个等式有多个解，如 $x = 0$ 、 $x = 1$ 等。建立在经典逻辑上的非确定性语义认为所有的这些解对系统来说是合法的。但确定性语义则将拒绝该模型。在 SR 语义中，这个等式有的解是 $x = \perp$ ，即 unknown，因此该模型被拒绝。

图 5-9 中右边的模型则可以表示为：

$$x = \neg x$$

符号 \neg 表示逻辑非。这时，以经典逻辑来理解这个等式，这个等式根本没有解，与上式完全不同的情况。在确定性语义中该模型仍然被拒绝。在 SR 语义中，这个等式的解同样是 $x = \perp$ ，为未知。因此，被拒绝。

第三个例子，如图 5-11 所示，由 Malik (1994) 提出。这个模型可以被确定性语义接受

而被结构性语义拒绝。逻辑上, 这个 AND 门的输出总是 false, 因此发送给 Display 角色的输出应该与 Bernoulli 角色产生的输入值取反后相等。因此, 对所有可能的输入, 输出都相同。但是, 当 Bernoulli 角色的输出为 false 时, 模型会当作非结构性 (non-constructive) 而被 Ptolemy II SR 指示器认为是非法的。这时, 回路中的所有信号保持为未知。在结构性 SR 语义中, 尽管有一个唯一的带有未知的定点, 该具有未知的解是最小固定点。因此, 模型的行为仍然是不确定的。

虽然图 5-11 中的电路对每一个输入似乎都有一个逻辑上一致的行为, 但是仍然可以找到充分的理由来拒绝它。如果把该模型实现为一个电路, 那么逻辑门之间带来的时延将导致电路产生振荡。事实上, 该模型的逻辑描述是无法实现的。如果用软件实现这些电路, 那么找到唯一定点并证明它是唯一的已知方法是穷举搜索, 即对所有的值进行尝试。在规模较小的模型中, 穷举搜索是可行的。但是, 对于规模较大的模型, 问题就会变得非常复杂。因为如果某个数据类型有无穷多个可能的值, 那么通过穷举搜索这种方法就无法找出唯一的固定点。因此, 该模型充分揭示了非结构性模型在实践上的困难。

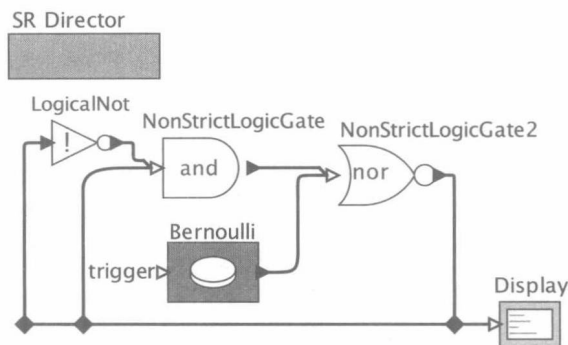


图 5-11 有唯一固定点的非结构性例子

现在, 简单地介绍 SR 语义理论基础。这其实是一个非常深奥的话题, 此处只作简单介绍。SR 语义基于完全偏序集 (CPO) 上的连续函数理论 (见第 5 章补充阅读: CPO、连续函数和固定点)。就 SR 来说, 关键 CPO 也叫作扁平的 CPO, 如图 5-12 所示。这个 CPO 由最小元素 \perp 和所有 Ptolemy 模型中的“合法的”值组成, 例如, 布尔值、整数、实数、元组、记录、列表等 (详见第 14 章)。在这个 CPO 序列中, 任何合法的值都比 \perp 大, 但是合法的值之间是不能相互比较的, 这就产生了术语“扁平的”。

现在, 思考这样一个 SR 模型: 模型中每个角色的输出都可以看成在上述扁平的 CPO 中取值的变量, 由所有输出变量组成的矢量可以看成在乘积 CPO 中取值的变量, 这个乘积 CPO 是在构建所有按



图 5-12 扁平的 CPO 保证 SR 中唯一最小固定点的存在

元素指向系列的 CPO 的笛卡儿乘积时产生的。为了使问题简单, 假设这个模型是闭合的, 这意味着模型中每个输入端口都与一个或者多个输出端口相连 (这一理论在开放模型中同样成立, 但是要更复杂一些, 读者可以在 Edwards 和 Lee (2003b) 中看到更详细的介绍)。这个模型定义一个函数 F , 该函数的定义域和上域都是这个乘积 CPO。因为这个模型是闭合的, 所以每个输入对应一个输出。因此, F 的输入为矢量 \bar{x} , 输出为矢量 \bar{y} , 矢量 \bar{y} 由模型中的角色点火产生。根据上述理论, 一个闭合的 SR 模型可以表示为以下等式:

$$\bar{x} = F(\bar{x})$$

这个等式有唯一的最小解 \bar{x}^* , 假设 F 是单调的 (monotonic), 即如果 $\bar{x} \leq \bar{y}$, 那么 $F(\bar{x}) \leq F(\bar{y})$ 。(这一结论成立的条件是函数是连续的, 但是在扁平的 CPO 中, 单调性函数就是连续的。) 因为 \bar{x}^* 满足 $\bar{x}^* = F(\bar{x}^*)$, 所以解 \bar{x}^* 称为固定点。而“最小的”也就意味着在 CPO 排序中 \bar{x}^* 比这个等式的其他解都小。也就是说, 对于其他任意的满足 $\bar{y} = F(\bar{y})$ 的 \bar{y} ,

一定有 $\vec{x}^* \leq \vec{y}$ 。

另外, 最小定点也可以通过有限次的迭代计算得出, 事实上, 迭代的次数最多为 N 次, N 是这个模型所有输出的个数。开始时所有的输出都设置为 \perp , 每次迭代都会点火那些没有到达定点的角色以保证其至少更新一个输出。一旦一个输出被更新, 它的值就从 \perp 变成一个合法的值 v 。因为 F 是单调的, 所以那些已经变成了 v 的输出的值不可能变成 \perp 或者其他 v' , 因为 $v > \perp$ 且 v 与其他 $v' \neq v$ 的 v' 都是不可比较的。因此, 每个输出的值最多可以更新一次。因此, 最多经过 n 次迭代, 一定可以求得最小定点。

只有每个独立角色都是单调的, 函数 F 的单调性才能得到保证; 也就是说, 点火函数是单调的。这也说明, 如果组成一个函数的每个部分都是单调的, 那么由这些单调部分组成的函数也是单调的。而 Ptolemy 中原子角色的单调性是采用结构性保证的。满足上述情况的关键是确保有一个角色, 如果有 unknown 输入时, 角色输出一个 v , 那么当这些输入变成 known 时, 角色将不会改变它的输出, 而使它的值变成 v' 。保证这一性质的一个简单方法是保证角色是严格的。在这个意义上, 这个角色就需要所有的输入都必须是 known, 否则它就会产生 unknown 输出。Ptolemy 中的大部分角色都是严格的, 但是前面讨论过的某些关键角色是非严格的。在 SR 模型中, 每个循环都需要一些非严格的角色。

补充阅读: CPO、连续函数和定点

SR 语义基于次序理论, 这里将进行简单介绍, 更详细的介绍可以参考 Davey and Priestly (2002)。

考虑一个集合 S , S 上的一个二元关系 (binary relation) 是一个子集 $\sim \subseteq S \times S$ 。常记作 $x \sim y$ 来代替 $(x, y) \in \sim$ 。 S 上的偏序 (partial order) 是一种二元关系 \sqsubseteq , 它具有自反性 (reflexive) (即 $\forall x \in S: x \sqsubseteq x$)、反对称性 (antisymmetric) (即 $\forall x, y \in S: x \sqsubseteq y, y \sqsubseteq x$, 则 $x=y$)、以及传递性 (transitive) (即 $\forall x, y, z \in S: x \sqsubseteq y, y \sqsubseteq z$, 则 $x \sqsubseteq z$)。一个偏序集是一个具有偏序性质的集合。

令 $X \subseteq S$, X 的一个上界是 u , $u \in S$, 所以有 $\forall x \in X: x \sqsubseteq u$ 。 X 的最小上界 (least upper bound) 记为 $\sqcup X$, 是一个元素 $\ell \in S$, 使得对于 X 的所有上界 u , 有 $\ell \sqsubseteq u$ 。 S 的链 (chain) 是一个子集 C , $C \subseteq S$, 它是全序的 (totally ordered): $\forall x, y \in C: x \sqsubseteq y$, 或者 $y \sqsubseteq x$ 。如果 S 是一个完全偏序集 (complete partial order 或 CPO) 同时也是一个偏序集, 那么 S 的每个链在 S 上都有一个最小上界。这个条件也保证每个 CPO S 都有一个底元素 (bottom element) \perp , 使得 $\forall x \in S: \perp \sqsubseteq x$ 。(一个空的链必须在 S 内有一个最小上界, 并且 S 的空子集的上界的集合就是整个 S 。)

为了更好地理解上面的概念, 思考这样一个自然数集合 $\mathbb{N} = \{0, 1, 2, \dots\}$, \mathbb{N} 是一个有着正常 (全序, 因此也是偏序) 顺序 \leq 的偏序集。因为 \leq 是全序, 所以 \mathbb{N} 是一个链。 \mathbb{N} 最小上界可以定义为一个新的数 ω , 使得对于所有的 $n \in \mathbb{N}$ 都有 $n < \omega$ 。 ω 不是自然数, 因此 \mathbb{N} 不是 CPO。另一方面, 集合 $\mathbb{N}^\omega = \mathbb{N} \cup \{\omega\}$ 是 CPO。 \mathbb{N}^ω 的底元素是 0。

链是有限的每个偏序集都是 CPO。这是因为链中的最大元素同时也是这个链的最小上界。这也是图 5-12 中的扁平偏序集是 CPO 的原因。

假设 X 和 Y 是两个 CPOS。如果所有的链 C 满足 $C \subseteq X$, $f(\sqcup C) = \sqcup \{f(c) \mid c \in C\}$, 那么函数 $f: X \rightarrow Y$ 是 Scott 连续的 (Scott-continuous) 或者简单连续的 (continuous)。如

果知道每个连续函数都是单调的 (monotonic), 即如果 $\forall x, y \in S: x \sqsubseteq y$, 则 $f(x) \sqsubseteq f(y)$ 。但是, 并不是所有单调的函数都是连续的。例如, 函数 $f: \mathbb{N}^\omega \rightarrow \mathbb{N}^\omega$, 使得对于所有的 $n \in \mathbb{N}$ 都有 $f(n)=0$ 和 $f(\omega)=\omega$ 。所以, $f(\sqcup \mathbb{N})=f(\omega)=\omega$, 而 $\sqcup \{f(n) \mid n \in \mathbb{N}\}=\sqcup \{0\}=0$ 。定点定理 (fixed-point theorem) 是次序理论的一个重要结论: 1) CPO X 上的每个单调函数 $f: X \rightarrow X$ 都有一个最小定点 x^* ; 2) 假设 f 是连续的, 那么 $x^* = \bigcup_{i \geq 0} f^i(\perp)$, 其中 $f^0(\perp) = \perp$, $f^{i+1}(\perp) = f(f^i(\perp))$ 。结论 2) 经常用于寻找一个有效的程序来计算 SR 模型的语义。

5.5 小结

本章介绍了 Ptolemy II 中的 SR 域。在 SR 中, 角色的执行由逻辑时钟决定, 从概念上来说, 在每一个时钟节拍, 所有的角色会同时且瞬时地执行。本章同时也介绍了 SR 模型是怎样产生定点的, 并以循环和非循环的 SR 模型为例进行了说明。另外, 还介绍了 SR 中允许不同部分响应不同速率时钟的情况。最后, 简单介绍了 SR 模型背后的数学理论基础。

练习

- 本练习的目地主要是熟悉 SR 中的 absent 事件的使用。
 - 作为热身, 请使用 Sequence 角色和 When 角色来构建一个产生带有 absent 的 true 值序列的 SR 模型。例如: (true, absent, absent, true, absent, true, true, true, absent) 必须确保模型能够将输出展示出来。特别是, absent 是可显示的^①。
 - 使用 Default 和 When 来创建一个复合角色 IsAbsent。该角色对于任意给定的输入序列, 在每个时钟节拍, 当输入为 absent 时, 输出为 true; 否则, 输出为 absent^②。
 - 创建一个复合角色, 它能够区分鼠标的单击与双击。这个角色必须有一个名为 click 的输入端口以及两个输出端口: singleClick 和 doubleClick。当在一个 true 输入后跟着的是 N 个 absent 时, 该角色必须在 singleClick 产生一个 true 值。 N 是这个角色的一个参数。类似地, 当在输入接收到了两个 true 信号后, 后面的输入为 N 个 absent 时, 这个角色应该在 doubleClick 产生一个 true 值。
如果在输入端口 click 的 N 个时钟节拍内在有 3 个 true 时, 模型该怎样运行的?
 - (可选练习) 用 Java 写一个自定义角色的方法重新实现上面 (a) ~ (c) 的功能。同使用原语 SR 角色实现的进行比较, 看看哪一个更加容易理解, 哪一个更加复杂?
- 假设 Arbiter 的每个实例都初始化了它的令牌 (将它的 initiallyOwnsToken 参数设置为 true), 那么图 5-6 的令牌环模型是结构性的 (constructive)。如果 Arbiter 的实例没有初始化它的令牌, 那么这个模型仍然是结构性的吗? 如果是, 说明为什么? 如果不是, 给出一系列 Request 角色的值, 使得模型出现因果循环。

① 注意使用 TrueGate 将更容易实现这个功能, 但是本练习的目的是让读者充分地了解 When。

② 这道题同样有一个更简单的实现方法, 那就是使用 IsPresent。但是, 这道题的目的是让读者充分地掌握 Default 的用法。

有限状态机

Thomas Huining Feng、Edward A. Lee、Xiaojun Liu、Christian Motika、
Reinhard von Hanxleden 和 Haiyang Zheng

有限状态机常用于给各种工程和科学应用的系统行为建模。系统的**状态**（state）是指它在特定时间点的所处的状况。**状态机**（state machine）是一个系统，它的输出不仅取决于当前的输入，而且还取决于系统当前的状态。系统的状态是对输出所需要的前期输入的一个总结。状态机由状态变量 $s \in \Sigma$ 表示，其中 Σ 是这个系统所有状态的一个集合。**有限状态机**（Finite State Machine，FSM）是 Σ 为有限集合的状态机。在有限状态机中，系统行为是由状态集，以及管理这些状态转移的规则共同建模而成的。

Ptolemy II 中有些角色包括了状态并且表现为简单的状态机。例如，Ramp 角色（它的功能是产生一个计数序列）具有状态，它的状态就是计数序列中当前的位置。这个角色使用一个用来跟踪记录当前值的本地变量，该本地变量称为**状态变量**（state variable）。Ramp 对一个触发脉冲（trigger）输入的响应依赖于它之前被点火的次数，而这一次数将被状态变量记录下来。Ramp 角色所有可能的状态数取决于计数序列的数据类型：如果是 int，那么它有 2^{32} 个可能的状态；如果是 double，那么它有 2^{64} 个可能的状态；如果是 String，那么可能的状态数是无限的（此时 Ramp 不能用有限状态机表示）。

尽管 Ramp 角色的状态数可能非常大，但是从一个状态变为下一个状态的逻辑非常简单，这使得表示角色的行为变得很容易。相反，有些角色可能的状态数很小，但是使一个状态移动到另一个状态的逻辑相对复杂。本章将重点研究后者。

本章主要讨论 Ptolemy II 中有限状态机的设计、可视化和分析方法。第 8 章将这些方法扩展到构建模态模型（modal model），这些模型中的状态本身也是 Ptolemy II 的模型。

6.1 Ptolemy 中的 FSM 创建

Ptolemy II 有限状态机的创建方法与前面介绍的面向角色模型的创建方式类似，只是有限状态机的创建采用状态和转移而不是角色与连接 / 关系。**转移**（transition）表示从一个状态到另一个状态的运动，转移是否发生由条件（guard）决定，条件表达式规定了转移发生的条件。它也可指定输出动作（输出动作决定当转移发生时得到怎样的输出）以及赋值动作（赋值动作在转移发生的时候给参数赋值）。

Ptolemy II 中用于实现 FSM 模型的主要角色都是，它可在 Utilities 库中调用^①。一个 ModalModel（模态模型）包含一个 FSM，它是一个如图 6-1 所示的使用可视化符号描述的

① 也可以使用 FSMActor，该角色可以在 MoreLibraries → Automata 中找到，它更简单，也正是因为这个原因，它不支持 6.3 节和第 8 章中提到的模式细化。

状态和转移的集合。尽管 ModalModel 通常可以有任意数量的输入和输出端口，但在图 6-1 中的 ModalModel 有 2 个输入端口和 2 个输出端口。它有 3 个状态，一个是**初始状态** (initial state) (即图中的 initialState)，这个状态是模型开始执行时角色的状态。初始状态用加粗的圆角矩形表示。**终止状态** (final state) 用双线的圆角矩形表示 (关于终止状态，后面会详细介绍)，如图 6-2 所示。在 Vergil 中创建了一个 FSM 的过程。

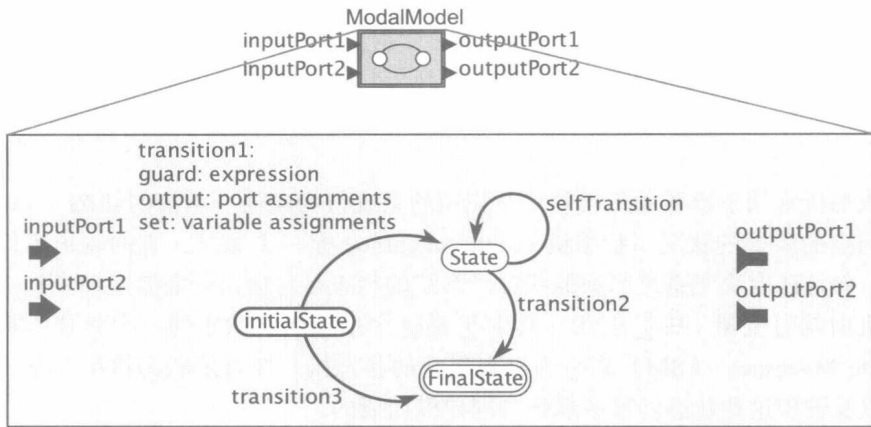


图 6-1 Ptolemy II 中状态机的表示符号

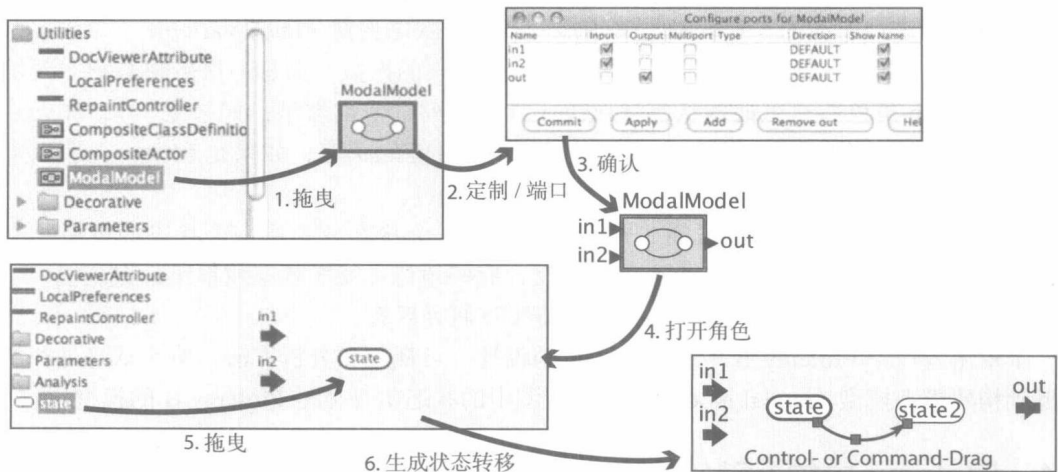


图 6-2 使用 ModalModel 角色在 Vergil 中创建 FSM (类似通过 FSMA ctor 创建一个程序应用)

创建一个 FSM，首先要从库中拖拽一个放到模型中。然后右击，选择 [Customize->Ports]，再单击 Add 并给输入或输出端口指定端口名，给角色添加输入与输出端口 (如果使用 Mac，可以按 control+ 单击)。然后右击，选择 Open Actor。图 6-3 展示了创建的结果窗口。它与 Vergil 中的其他窗口类似，但是它有一个主要由 State 组成的自定义库、一个参数 (parameter) 库以及一个用于对设计进行注释的装饰元素 (decorative element) 库。

拖入一个或更多的状态。按住 control 键 (如果使用的是 Mac，请按 command 键)，单击一个状态，然后拖向另一个状态，这样就创建了状态与状态之间的转移。通过拖动转移上的“控制点”，可以调整图中转移的位置与弧度。

双击转移 (或者右击，然后选择 Configure)，可在图 6-4 所示对话框的条件表达式

(guardExpression) 中设置条件 (guard)、输出动作 (outputActions) 和赋值动作 (setActions) 为了使模型具有更强的可读性，可以为转移添加相关注释 (annotation)。

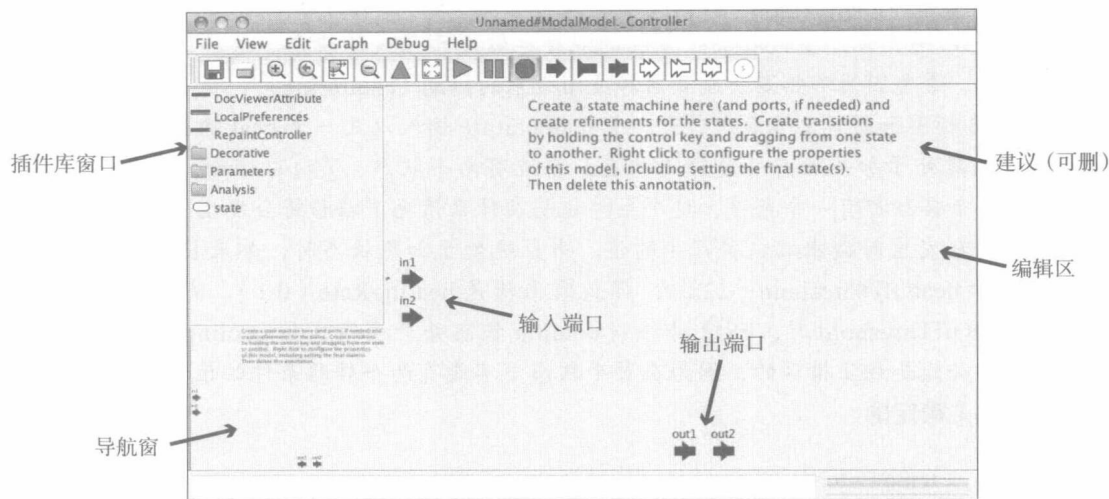


图 6-3 在 Verilog 中编辑 FSM，在 FSM 之前添加两个输入端口和两个输出端口

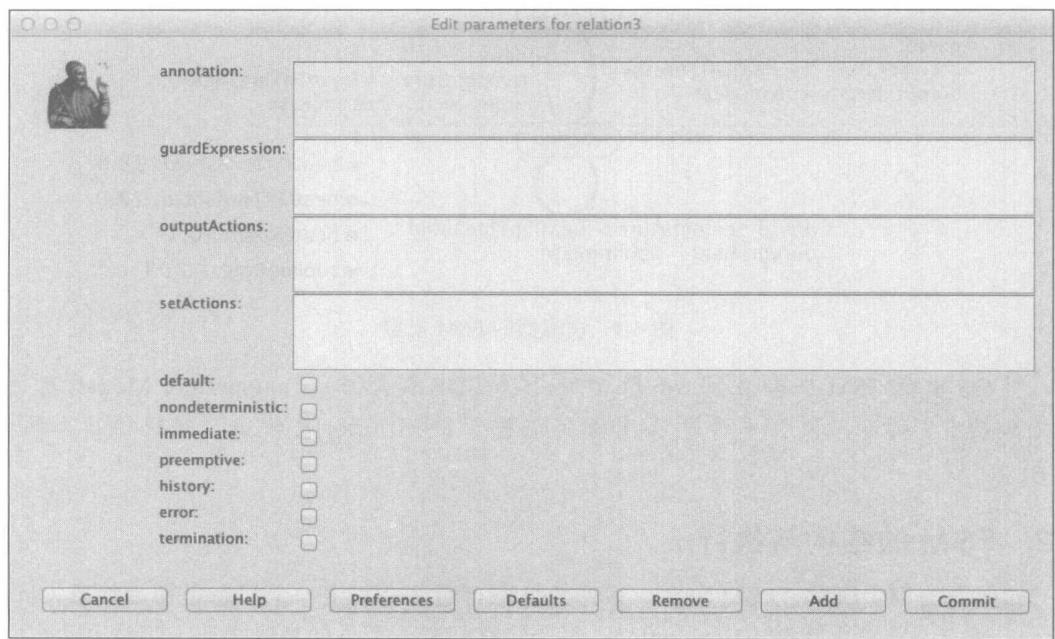


图 6-4 在 FSM 中设置转移的对话框

可以将实现的有限状态机放在更大的模型中，用另一个指示器执行。指示器的选择取决于应用。所有的指示器与该 FSM 都是兼容的。

下面这个例子详细解释了 FSM 应用的使用过程。

例 6.1 考虑一个控制加热器的恒温器。恒温器被建模为一个状态为 $\Sigma=\{\text{heating, cooling}\}$ 的状态机。如果状态 $s=\text{heating}$ ，那么加热器开启；如果 $s=\text{cooling}$ ，那么加热器停止运行。假设设定的温度是 20°C 。当温度高于或者低于目标温度时，加热器就在开与关之

间切换，这是不可取的。因此，状态机应该在设定值（setpoint）附近滞后（hysteresis）。如果加热器开启，恒温器允许温度慢慢地上升到稍微超过设定的目标温度 22℃，如果关闭加热器，恒温器允许温度下降到低于目标温度的下限 18℃。值得注意的是，系统的温度在 18℃~22℃之间变化，不仅取决于系统的温度也取决于系统的状态。这种策略可在温度接近设定值时，避免因为加热器快速开启和关闭带来的抖动（chattering）。

该 FSM 模型如图 6-5 所示。它有一个 temperature 输入以及一个 heat 输出，这个输出指明现在加热器处于加热状态还是停止状态。系统有两个状态， $\Sigma=\{\text{heating}, \text{cooling}\}$ ，有 4 个转移，每个转移都有一个条件，这个条件说明在什么情况下转移将会发生。同时，转移也说明了当转移发生时输出端口将产生的值。当系统处于加热状态时，如果 temperature 输入小于或等于 heatOffThreshold (22.0)，那么输出值是 heatingRate (0.1)。当 temperature 输入大于 heatOffThreshold 时，FSM 转移到 cooling 状态并产生输出值 coolingRate (-0.05)。注意，条件必须是相互排斥的，因为在每个状态下不能有两个转移条件的值为 true，这才能保证状态机是确定的。

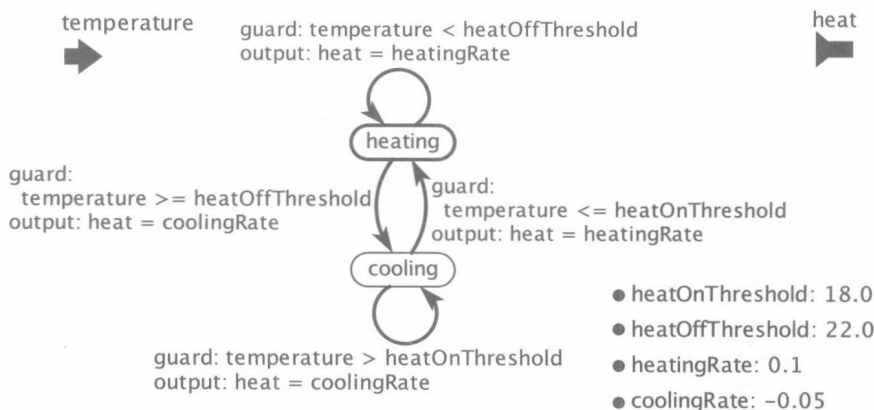


图 6-5 恒温器的 FSM 模型

图 6-5 中的 FSM 嵌套在图 6-6 所示的一个 SDF 模型中。Temperature Model 角色的定义如图 6-7 所示，模型中外界温度的变化基于 FSM Actor 的输出。系统输出如图 6-8 所示。

6.2 FSM 的结构与执行

如前面的例子所示，FSM 包含一系列的状态与转移。其中一个状态是初始状态（initial state），还有一些终止状态（final state）。每个转移都有一个条件表达式和一些输出动作以及赋值动作。在状态机开始执行时，角色的状态被设置为初始状态。每次角色的点火都要按照下面的步骤进行，在执行的点火阶段，角色将：

- 1) 读取输入。
- 2) 根据当前状态的传出转移进行条件计算。
- 3) 选择条件为 true 的转移。
- 4) 如果有转移发生，则根据选择的转移执行输出动作。

在后点火阶段，角色将：

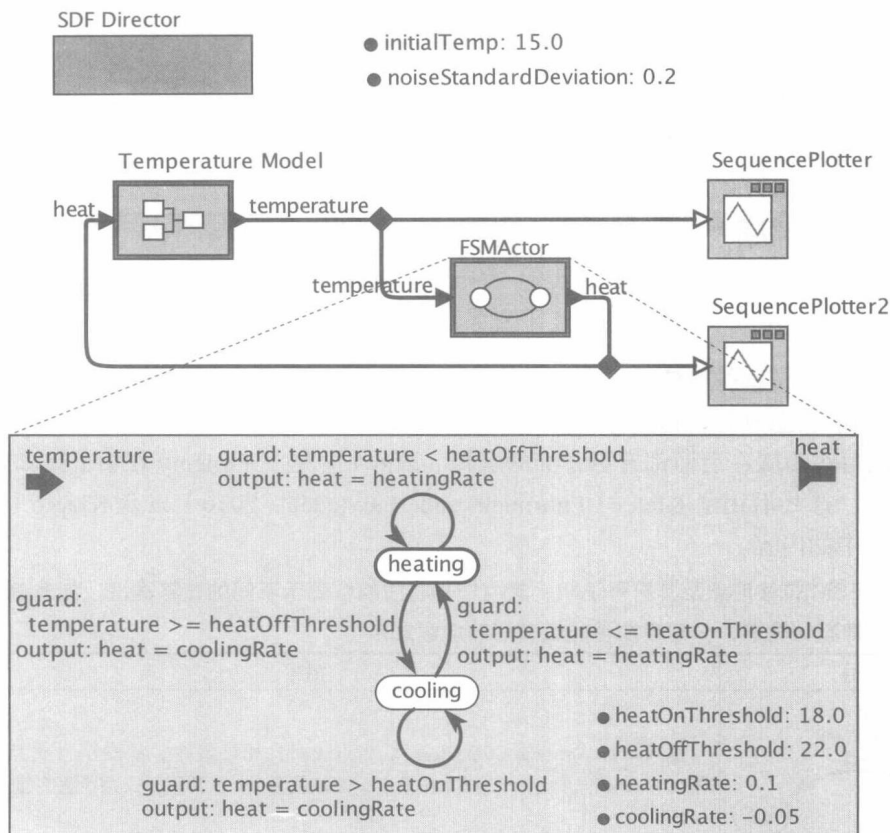


图 6-6 图 6-5 中的恒温器的 FSM 模型嵌入 SDF 模型中。Temperature Model 角色如图 6-7 所示

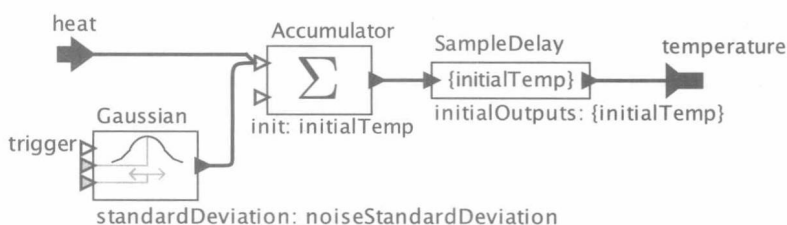


图 6-7 图 6-6 中的 Temperature Model 复合角色

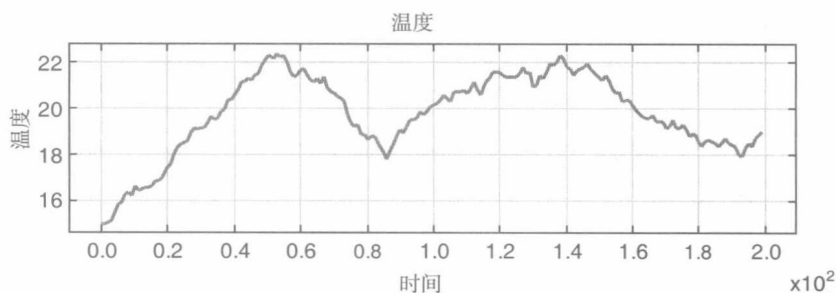


图 6-8 图 6-6 产生的两个图，上图表示温度，下图表示加热速率（heating rate）的值，这个值反应了加热器是处于加热状态还是关闭状态。两个图都是关于时间的函数

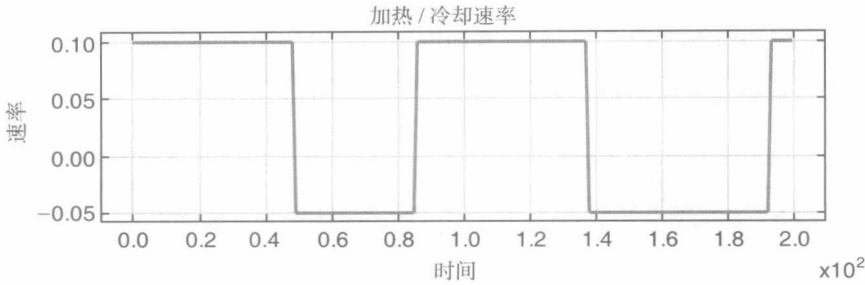


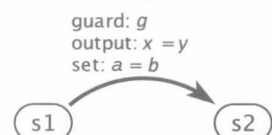
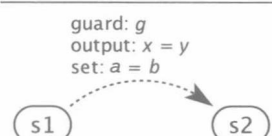
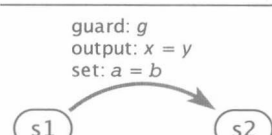
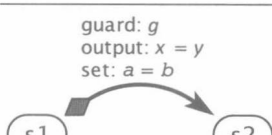
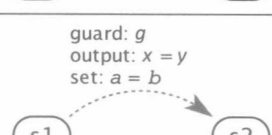
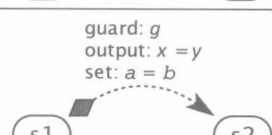
图 6-8 (续)

5) 执行选择转移的赋值动作 (set action)，给参数赋值。

6) 将当前状态改为转移选择的目标状态。

上面的每个步骤在后面都有更详细的解释。表 6-1 总结了 Ptolemy II 中 FSM 转移（无分层）的标记，这些沿用了 Kieler (Fuhrmann and Hanxleden, 2010) 以及 Klepto (Motika et al., 2010) 中的内容。

表 6-1 FSM 转移汇点及其符号说明，其中符号将被组合表示不同的转移类型。如非确定性，即默认转移，分别用灰色起始菱形和虚线表示

符号	描述
	普通转移 (ordinary transition)。当点火时，如果条件 g 为 true (或者没有条件)，那么 FSM 将选择这个转移并在输出 x 被赋值为 y 。在转移时，角色将变量 a 赋值为 b
	默认转移 (default transition)。当点火时，如果没有其他的非默认转移且条件 g 为 true，那么 FSM 角色将按照上面的规则选择这个转移，产生输出，采用与上面相同的方式给变量赋值
	非确定性转移 (nondeterministic transition)。这个转移允许在同一次迭代中使能另一个非确定性转移。将非确定性地选择其中一个使能的转换
	立即转移 (immediate transition)。如果状态 $s1$ 是当前状态，那么这个转移的作用与普通转移一样。但是，如果 $s1$ 是即将发生的某个转移的目的状态，且从 $s1$ 到 $s2$ 的条件 g 为 true，那么 FSM 将立即转移到 $s2$ 状态。即在一次迭代中发生了两次转移。在角色被点火时，输出 x 被赋值为 y ，在转移发生时变量 a 被赋值为 b 。如果立即转移链中不止一个转移给输出或者变量赋值，那么以最后一个转移的值为准
	非确定性默认转移 (nondeterministic default transition)。一个拥有默认转移优先级的非确定性转移，与其他非确定转移相比优先级更低
	立即默认转移 (immediate default transition)。一个拥有默认转移优先级更低的立即转移，与其他立即转移相比优先级更低

6.2.1 转移条件定义

给状态转移定义一个恰当的条件是创建有限状态机的重要工作内容。但是，正如下面要讨论的，指示器采用确定的模型执行方式，使得有些条件表达式可能得到不希望发生的结果。具体来说，不同的指示器会以不同的方式处理 `absent` 输入，这就使得条件表达式的计算是反直观的一种形式进行。

每个转移都有一个条件 (guard)，其依赖于状态机的输入、参数、变量并产生模式细化输出的断言。

例 6.2 在图 6-5 中，条件表达式为

```
temperature < heatOffThreshold,
```

其中变量 `temperature` 指的是端口 `temperature` 的当前值，`heatOffThreshold` 指的是一个名为 `heatOffThreshold` 的参数。

表 6-2 给出了一个具有输入端口 p 和 q 以及参数 a 的 FSM 中的几个有效条件的例子。

表 6-2 条件表达式的例子，其中 p 和 q 是端口， a 是参数

	条件	描述
1		表达式为空的条件值永远为 <code>true</code>
2	<code>p_isPresent</code>	如果端口 p 有令牌，那么条件的值为 <code>true</code>
3	<code>p</code>	如果端口 p 有令牌，且它的值为 <code>true</code> ，那么条件的值为 <code>true</code>
4	<code>!p</code>	如果端口 p 有令牌，且它的值为 <code>false</code> ，那么条件的值为 <code>true</code>
5	<code>p>0</code>	如果端口 p 有令牌，且它的值大于 0，那么条件的值为 <code>true</code>
6	<code>p>a</code>	如果端口 p 有令牌，且它的值大于参数 a 的值，那么条件的值为 <code>true</code>
7	<code>a>0</code>	如果参数 a 的值大于 0，那么条件的值为 <code>true</code>
8	<code>p&&q</code>	如果端口 p 和 q 有令牌，且它的值都为 <code>true</code> ，那么条件的值为 <code>true</code>
9	<code>p q</code>	如果端口 p 和 q 的输入中有一个值为 <code>true</code> ，那么条件的值为 <code>true</code>
10	<code>p_0>p_1</code>	如果端口 p 在通道 0 和信道 1 都有令牌，且通道 0 的令牌要大于通道 1 的令牌，那么条件的值为 <code>true</code>
11	<code>p_1_isPresent&&(P_0 p_1)</code>	如果端口 p 在通道 0 和信道 1 都有令牌，且两个令牌中有一个为 <code>true</code> ，那么条件的值为 <code>true</code>
12	<code>timeout(t)</code>	如果在源状态的时间经过 t ，那么条件的值为 <code>true</code>

如表的第二行所示，对于任意的端口 p ，符号 `p_isPresent` 可以用在条件表达式中。如果端口 p 的输入令牌为 `present`，那么条件的值为 `true`。相反， p 为 `absent` 时，表达式 `!p_isPresent` 的值将为 `true`。注意，在域中， p 永远不会为 `absent`，例如 `PN`，这个表达式的值永远不为 `true`。

如果端口 p 没有输入令牌（它所有的通道都为 `absent`），那么表中除了第一个条件以外所有条件的值都是 `false`。具体地说，如果 p 是布尔类型且没有输入令牌，那么 `p` 和 `!p` 的值都可能是 `false`。类似地，`p > 0`、`!(p > 0)` 以及 `p <= 0` 也有可能为 `false`。当然，这种情况仅仅当这个 FSM 与一个指示器一起使用，而该指示器可以用 `absent` 输入激活这个 FSM，例如 `SR` 与 `DE` 域。

注意，由于 `absent` 输入可以被模型处理，所以对于表中的条件 9，含义会有细微的变化。此时该表达式只有 p 有输入令牌时，条件的值才有可能为 `true`，但是，不需要端口 q 有令牌。如果要求 p 和 q 两个端口都有输入令牌，这个转移才有可能发生，那么条件应该写成

```
q_isPresent && (p || q)
```

如果要使这个表达式更清晰（实际上，不必要这么做），可以写成

```
(p_isPresent && q_isPresent) && (p || q)
```

简言之，如果端口 p 为 **absent**，任何包含输入端口 p 的条件表达式的计算值都为 **false**。但是，如果子表达式中的 p 值没有被计算，那么条件的值也可能不会计算为 **false**。尤其是，当左边的表达式为 **true** 时，逻辑或（符号为 $||$ ）将不会对右边的内容进行计算。这也就是为什么条件 9 中的 q 的值为 **absent** 也可以使条件的值为 **true** 的原因。

这种计算策略的结果就是错误的条件表达式可能没有被检测到。例如，如果定义一个条件表达式 $p.foo()$ ，但 $foo()$ 在 p 的数据类型中没有定义，那么当 p 为 **absent** 时，这个错误将无法被检测出来。 p 出现的表达式都被计算为 **false**，不管 p 是否有令牌，表达式 $true || p < 10$ 的计算永远为 **true**。

对于具有多通道的多端口，条件表达式可以使用 p_i 来表示第 i 个通道，其中 i 是 $0 \sim n-1$ 的整数， n 是连接到端口的通道数。例如，表 6-2 中的第 10 行就有来自同一个输入端口的两个通道的输入令牌。类似地，第 11 行的条件表达式用到了 $p_i_isPresent$ 。

第 12 行的条件表达式表示经过一段时间后触发某个转移。表达式 $timeout(t)$ 中的 t 是一个 **double** 类型的变量，当 FSM 在源状态停留了 t 个单位时间后，这个表达式就判定为 **true**。在部分支持时间的域中，例如 SDF 与 SR，当 FSM 下次点火的时间大于或等于 t 时转移将会发生（当然，只有在指示器中的参数 **period** 不为 0 时转移才会发生），详见 3.1.3 节。在完全支持时间的域中，例如后面章节将讲到的 DE 与 Continuous，在进入源状态并经过 t 个时间单元后，转移就会立即发生，除非有其他的转移在 t 个时间单元以前就满足了转移的条件。

不论什么情况，输入端口或者参数的类型都必须与条件表达式中的类型相匹配。例如，如果端口 p 的类型为 **int**，那么表达式 $p || q$ 将会触发一个异常。

6.2.2 输出动作

一旦选择了一个转移，就执行它的输出动作（output action）。输出动作由转移的 **outputAction** 参数描述（见图 6-4）。输出动作的格式为 $portName = expression$ ， $expression$ 可能与输入值（就像条件表达式一样）或者某个参数有关。例如，在图 6-5 中，

```
output: heat = coolingRate
```

表示名为 **heat** 的输出端口的值由参数 **coolingRate** 给出。

下一节介绍了两种状态机，（见第 6 章补充阅读：状态机模型）Mealy 机和 Moore 机。上面所描述的行为构成了 Mealy 状态机，通过对产生输出的状态细化来实现 Moore 机，如第 8 章所述。

可以通过分号将多个输出动作分开，如 $port1 = expression1; port2 = expression2$ 。

6.2.3 赋值动作和扩展有限状态机

转移的赋值动作（set action）可以用来设置状态机参数的值。这一特性可以用来创建扩展状态机（extended state machine），它是一个具有数值状态变量的有限状态机。称为“扩展后的”，因为状态的数量取决于变量可取的不同值的个数，它甚至是无限的。

补充阅读：状态机模型

在文献中经常将状态机描述为五元组 $(\Sigma, I, O, T, \sigma)$ 。 Σ 是状态的集合， σ 是初始状态。

非确定性状态机可能有不止一个初始状态,在这种情况下 σ 本身就是一个集合,并且 $\sigma \subset \Sigma$, 尽管 Ptolemy II FSM 并不支持这一特殊情况。 I 是输入可能值的集合。在 Ptolemy II 的 FSM 中, I 是一个函数集,这些函数的形式都为 $i:P_i \rightarrow D \cup \{\text{absent}\}$, 其中 P_i 是输入端口集合 (或者说输入端口名字的集合), D 是在每一次点火时输入端口可能出现的值的集合, absent 表示输入 “absent” (也就是说, 当 $p_isPresent$ 判定为 false 时, $i(p) = \text{absent}$)。类似地, O 是每次点火时输出端口所有可能值的集合。

对于确定性状态机, T 是 FSM 中转移关系的函数, 这些转移都表示为 $T:\Sigma \times I \rightarrow \Sigma \times O$ 。事实上, 条件与输出动作只是这个函数的编码。对于非确定性状态机 (Ptolemy II 支持的), T 的上域是 $\Sigma \times O$ 的幂集, 允许有多个目标状态和输出值。

状态机的经典理论 (Hopcroft 和 Ullman, 1979) 使 Mealy 型状态机和 Moore 型状态机之间有明显的区别。输出只和状态有关而与输入无关称为 Moore 机, 输出与状态和输入都有关则称为 Mealy 型状态机。Ptolemy II 支持这两种模型, 它使用输出动作来实现 Mealy 型状态机, 使用模态模型中的状态细化来实现 Moore 型状态机。

Ptolemy II 状态机事实上都是扩展后的状态机, 它需要要比上面给出的模型更丰富的模型。扩展后的状态机增加了一个变量值集合 V , 它们的函数形式为 $v:N \rightarrow D$, 其中 N 为变量名的集合, D 是变量的取值集合。扩展状态机是一个六元组 $(\Sigma, I, O, T, \sigma, V)$, 其中转移函数的形式为 $T:\Sigma \times I \times V \rightarrow \Sigma \times O \times V$ (对于确定性状态机)。这个函数由转移、条件、输出动作和 FSM 的赋值动作组成。

例 6.3 图 6-9 显示了一个扩展状态机简单例子。在该例中, FSM 有一个名为 `count` 的参数。在赋值动作中从初始状态 `init` 到 `counting` 状态的转移将 `count` 赋值为 0。`counting` 状态有两个传出转移, 一个是自转移, 另一个是指向 `final` 状态的转移。当 `count` 小于 5 时, 自转移就会发生。通过赋值动作这个转移使 `count` 的值加 1。当 `count` 的值小于 5, 将不断点火, 直到 `count` 等于 5, 状态从 `final` 转为输出 5。在后续的点火中, 由于 `final` 状态的自循环, 所以输出都为 5。因此, 这个模型输出序列为 0, 1, 2, 3, 4, 5, 5, 5...

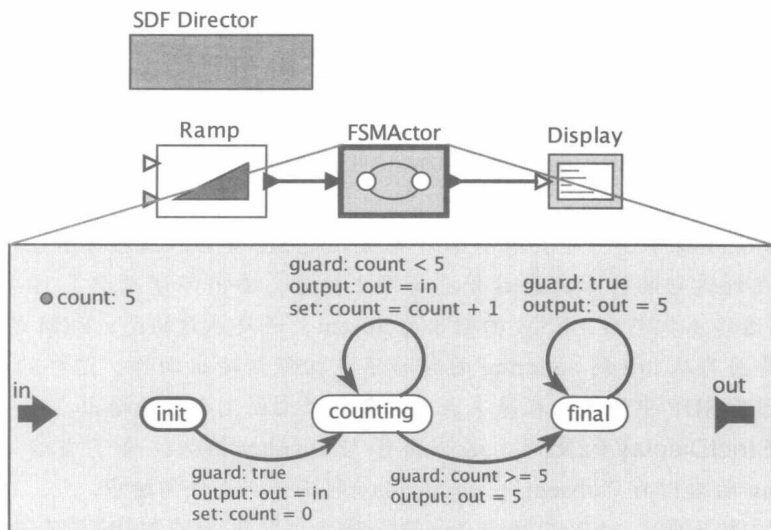


图 6-9 扩展的状态机, 变量计数器 (`count`) 是这个系统状态的一部分

6.2.4 终止状态

FSM 可能有终止状态 (final state)，进入终止状态后就意味着状态机的执行结束。

例 6.4 图 6-10 显示了例 6.3 的一个变化形式。变化主要表现在将 final 状态的 `isFinalState` 参数赋值为 `true`，图中由双线圆角矩形表示。一旦进入该状态，FSM 就会告知上层指示器它不希望再次被执行（通过它的后点火函数返回 `false` 使得模型不再继续执行）。因此，发送给 Display 角色的输出是有限序列 0, 1, 2, 3, 4, 5, 5。注意，最后的两个 5。这强调了一个事实——条件的计算在赋值动作之前。因此，在第 6 次点火开始，FSM 的输入是 5，count 的值为 4。此时 counting 状态的自循环就要发生，产生输出 5。在下次点火开始，count 为 5，所以执行指向 final 状态的转移，并产生另一个值为 5 的输出。

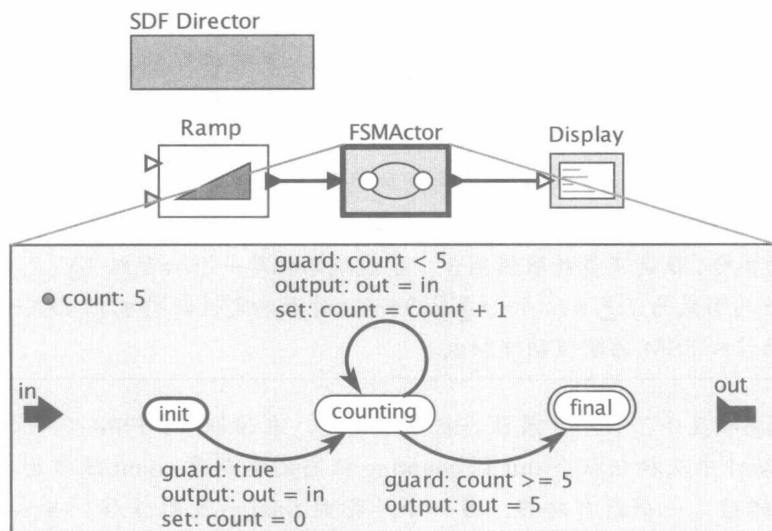


图 6-10 有终止状态的状态机，终止状态意味着状态机执行的结束

在 FSM 进入终止状态的迭代时，ModalModel 或者 FSM 角色的后点火 (`postfire`) 函数将返回 `false`。它将告诉围合上层指示器，FSM 不希望再次被点火，大多数指示器将不再点火 FSM，但是会继续执行这个模型的剩余部分。但是，SDF 指示器不同，因为它假设所有的角色同时获得输入并产生输出，而且它以静态的方式构建调度方式，另外它也不能调度非点火的角色。因此，如果任意角色的后点火 (`postfire`) 返回 `false`，那么 SDF 指示器将停止该模型的执行。相反，SR 指示器将继续执行，但是现在终止的 FSM 的所有输出在后续的时钟节拍最终都将是 `absent`。

例 6.5 图 6-11 显示了一个具有有限计数功能的 SR 模型，但它与例 6.4 中模型不同的是，这个模型在到达它的终止状态后并没有停止执行。输出部分显示了 10 次迭代的结果。注意在 FSM 到达终止状态后，FSM 的输出为 `absent`，还要注意的，FSM 的第一个输出也为 `absent`。这是因为从 `init` 到 `counting` 的转移不包括任何输出动作。这样的转移在 SDF 中是不兼容的，因为 SDF 中的角色在每次点火时都要产生固定个数的输出。

注意 `NonStrictDisplay` 的使用。这个角色与 `Display` 相似，除了当输入为 `absent` 时，`NonStrictDisplay` 的输出为 “`absent`”，而 `Display` 则不会产生任何输出。

正如上面的例子所述，SR 支持 `absent` 这一概念。数据流域和 PN 没有这个概念。产生输出失败将使得下游的角色无法得到输入从而无法执行。在 PN 中带有终止状态的 FSM 在

它到达终止状态时将停止产生输出。如果导致饥饿 (starvation)(即, 其他角色需要来自 FSM 的输入才继续执行), 那么整个模型就会终止运行。

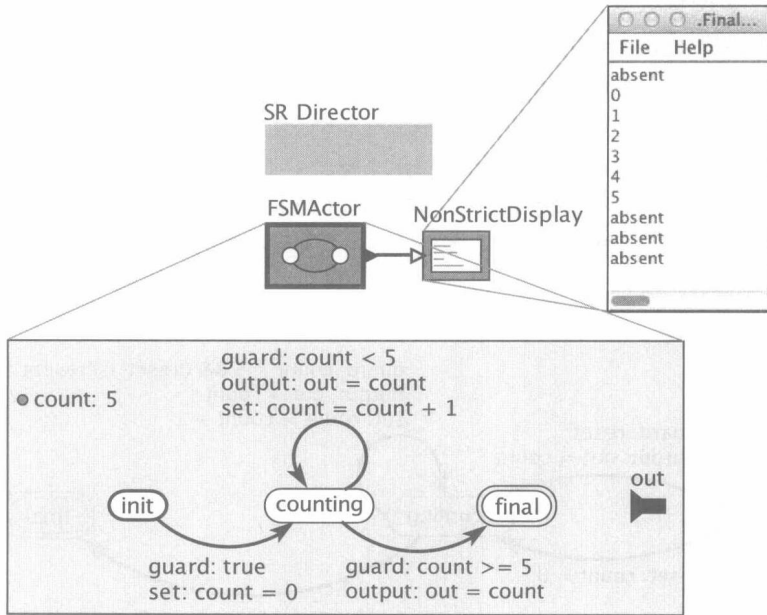


图 6-11 SR 模型中带有终止状态的状态机

6.2.5 默认转移

FSM 可有默认转移 (default transition), 这需要在创建转移时将 default 参数设置为 true (见图 6-4)。这些转移在其他非默认的传出转移 (outgoing) 都不满足执行条件时就会自动执行。默认转移使用虚线表示。

例 6.6 图 6-5 中的恒温器 FSM 可以采用图 6-12 所示的默认转移等效地实现。这里, 如果指向其他状态的传出转移不满足执行条件, 那么默认转移就会执行, 然后 FSM 将停留在同一状态并产生输出。

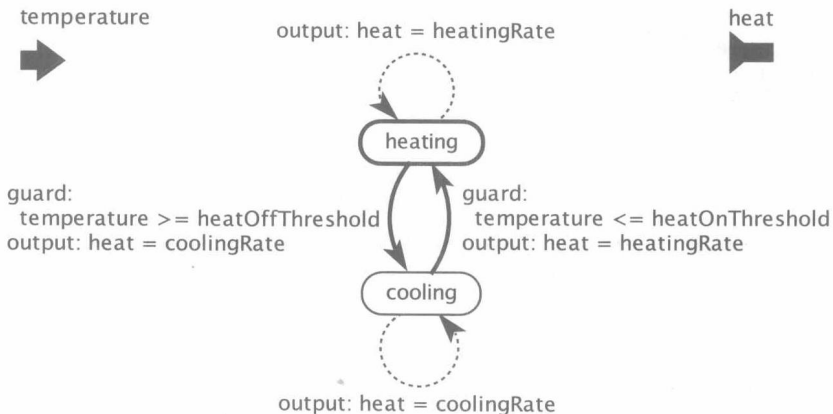


图 6-12 等效于图 6-5 的 FSM, 它使用默认自转移 (图中表示为虚线)。如果都不满足其他传出转移条件, 那么默认转移就会执行

如果默认转移也有条件表达式，那么这个转移只有在这个条件的值为 `true` 且没有其他非默认转移可以执行时才可以执行。因此，默认转移提供了**优先级**（priority）的一个基本形式，非默认转移的优先级大于默认转移。与某些状态机语言（例如，SyncCharts）不同，Ptolemy II FSM 提供了 2 个优先级，尽管它可以使用条件来设置任意级数的优先级。注意，使用计时计算模型的默认转移有一定的难度，详见 8.5 节。

默认转移经常用于简化条件表达式，下面一个例子就是。

例 6.7 考虑例 6.5 的计数状态机的例子，如图 6-11 所示。增加了一个 `reset` 输入，如图 6-13 所示，使得计数器可以被重置。如果 `reset` 的值为 `true`，那么状态机就会回到 `init` 状态。但是，图 6-13 的实现必须有所修改。`counting` 状态的两个传出转移必须添加以下条件

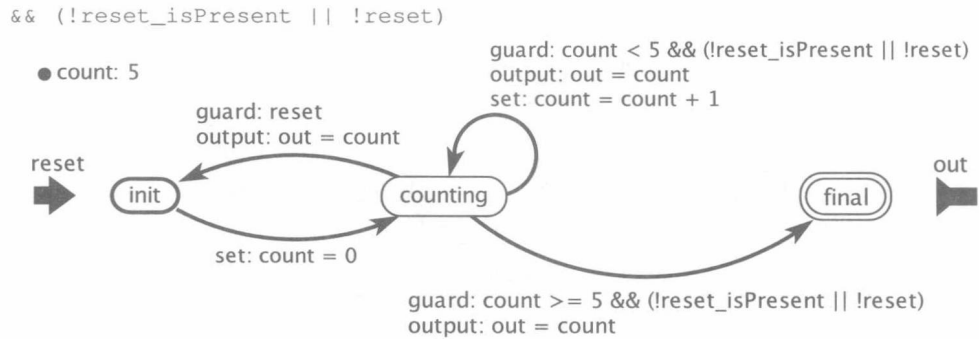


图 6-13 类似于图 6-11 的状态机，但是有一个额外的 `reset` 输入端口

这条语句保证 `counting` 状态的自转移只有在 `reset` 输入为 `absent` 或者 `false` 时才会发生。没有这条语句，这个状态机就是非确定性的，因为 `counting` 状态的两个传出转移可能同时执行。但是，这条功能简单的语句使得条件表达式看起来非常复杂。图 6-14 显示了使用默认转移实现的版本，这就意味着状态机只有在 `reset` 输入不是 `present` 或者不为 `true` 时，状态机才会执行计数功能。

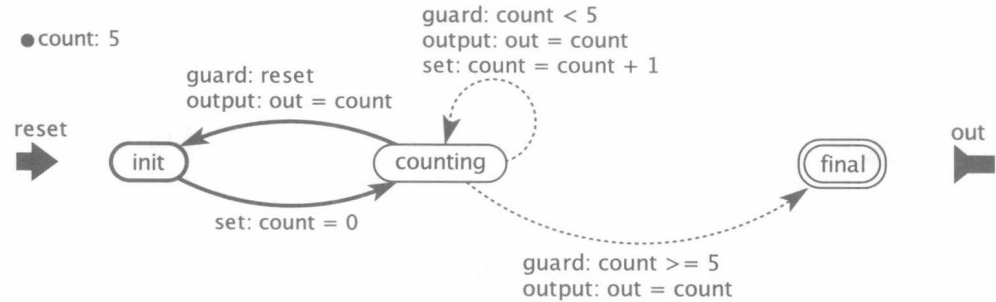


图 6-14 使用默认状态简化条件表达式的类似于图 6-13 的状态机

对于这个状态机，如果在第 4 次点火时 `reset` 是 `present`，那么开始的几个输出为 `absent`, 0, 1, 2, `absent`, 0, 1。在迭代中，当 `reset` 是 `present` 且为 `true` 时就会产生输出“2”，然后状态就重新开始。

6.2.6 非确定性状态机

如果在某个时刻不止一个条件的计算值为 `true`，那么这个 FSM 就是一个非确定性有限

状态机 (nondeterministic FSM) (除非其余的条件都是默认转移条件)。同时满足执行条件的转移叫作非确定性转移 (nondeterministic transition)。默认情况下, 转移必须是确定性的, 因此如果不止一个条件的值为 true, Ptolemy 将抛出如下异常:

```
Nondeterministic FSM error: Multiple enabled transitions found but
not all of them are marked nondeterministic.
in ... name of a transition not so marked ...
```

然而, 在某些情况下, 也允许非确定性转移。因为非确定性转移为那些需要对同一输入呈现不同形为的系统提供了一个很好的模型。非确定性转移对于构建可能的故障 (可能没有故障信息出现) 模型非常有用。要创建非确定性转移, 只需将图 6-4 中的 nondeterministic 参数设置为 true, 这样即使有其他转移满足执行条件, 该转移也满足执行条件。

例 6.8 图 6-15 显示了一个有故障的恒温器模型。当 FSM 在 heating 状态时, 两个传出转移都满足执行条件 (它们的条件的值都为 true), 因此任意一个转移都能执行。两个非确定性转移在图中用深灰色表示。图 6-16 是模型执行的结果。注意因加热器只能在短时间内开启, 使得温度在 18°C 左右徘徊, 该温度是加热器打开的阈值。

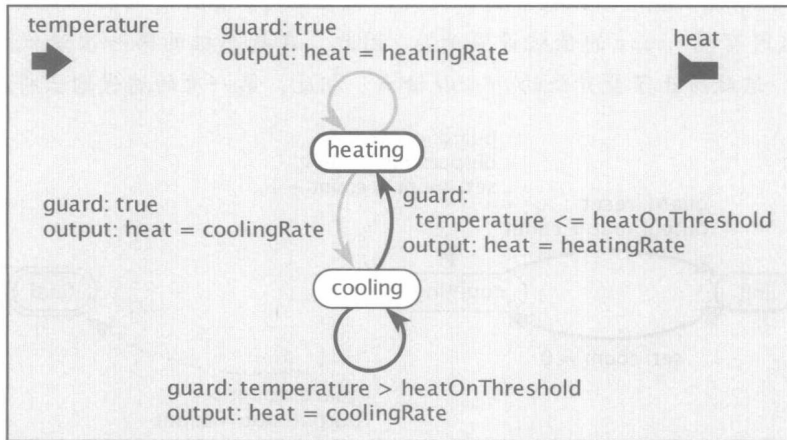


图 6-15 从 heating (加热) 到 cooling (冷却) 非确定性转移的有故障的恒温器模型

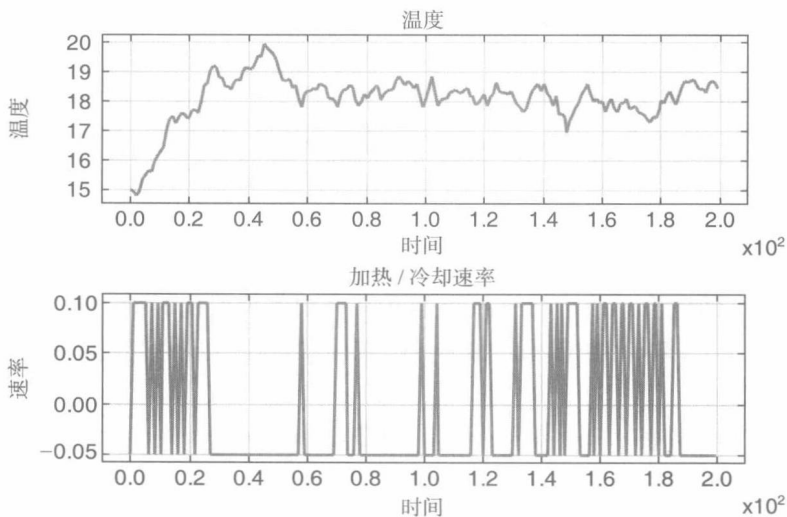


图 6-16 图 6-15 (图 6-5 的变体) 中的恒温器 FSM 的结果图

在非确定性 FSM 中，如果不止一个转移满足条件且它们都是非确定性转移，那么 ModalModel 或者 FSM 角色中的 fire 函数就会随机地选择一个转移执行。如果 fire 函数在一次迭代中不止调用一次（见 6.4 节），那么在同一次迭代中的后续调用将一直选用同一个转移。

6.2.7 立即转移

目前为止讨论都是 FSM 每次点火仅有一次转移的情况。但是，通过使用立即转移 (immediate transition) 也可能在一次点火中实现多次转移。如果状态 A 有一个转向另一状态 B 的立即转移，那么，如果立即转移的条件为 true，一旦状态转移到 A ，立即转移会在同一次点火中执行。在同一次点火中，状态 A 的传出转移和传入转移都将执行。这种情况下，状态 A 就称为过渡态 (transient state)。

例 6.9 在例 6.7 中，恒温器在第一次迭代中的输出是 absent 并且立即跟着 reset。如图 6-17 所示，可以通过将 init 到 counting 之间的转移设置为立即转移来避免 absent 输出。这样的改进有两个好处。第一，当模型初始化时，从 init 到 counting 的转移就会立即发生（在初始化时），这时变量 count 的值被设置为 0。因此，在状态机的第一次迭代后，它将处于 counting 状态。这就防止了最开始的 absent 输出。相反，第一次的迭代输出将为 0。

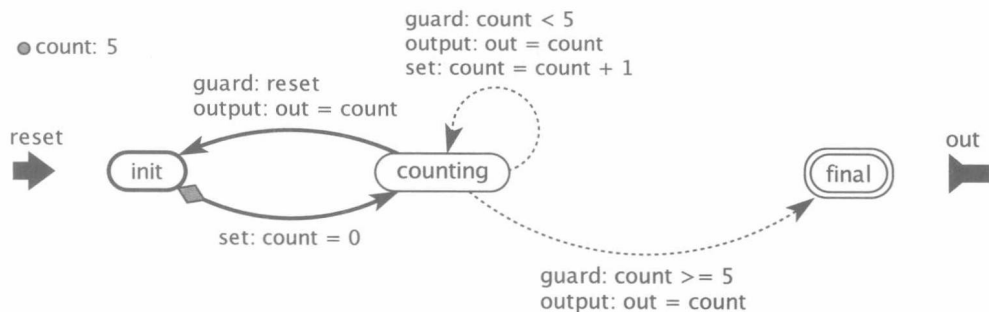


图 6-17 与图 6-14 中的状态机类似，但该状态机使用立即转移防止计数前输出 absent。立即转移由带深灰色菱形的箭头表示

第二，在 counting 状态时，如果 reset 输入为 present 且为 true，那么状态机将从 counting 转移到 init，并在同一次迭代中返回到 counting，将变量 count 设置为 0。

对于该状态机，如果在第四次迭代中 reset 为 present，那么前几个输出将是 0, 1, 2, 3, 0, 1。当 reset 为 present 且为 true 时，输出“3”由转移到 init 的转移产生，并且状态机重新开始。

注意，过渡态与状态机中停留时间为 0 的状态不一样。因为 Ptolemy II 中有超密时间 (superdense time) 的概念，所以状态机可能在某个状态停留的时间为 0，但是不同的微步 (microstep) 索引中，该状态的传入转移与传出转移发生在不同的点火时刻。（详见第 6 章进一步探索：弱暂态）。

当状态机响应时，响应的起始状态叫作现态 (current state)。现态可能发生立即转移。要让拥有立即转移的状态成为现态，就必须将转移条件前面的响应立即设置为 false，否则这个状态就会成为过渡态而不会成为现态。当现态既有立即又有非立即的传出转移时，状态

对这两类转移的处理是一样的。这两者之间没有明显的区别，它们在优先级上也是相同的。如果现态的立即传出转移和非立即传出转移条件的值都为 true，那么它们两者之间的任意一个应为默认转移（default transition）或者这两个转移都要设置为非确定性转移。

如果初始状态（initial state）有立即传出转移，那么当 FSM 初始化时它们就会计算条件的值。如果条件的值为 true，那么在 FSM 开始执行之前这个转移就会发生。注意在某些域中，例如 SR，在执行开始之前产生输出，这样输出将不会被目标状态所察觉，所以在这些域中，初始状态的立即转移将不会产生输出。

在某些时候，立即转移、默认转移以及非确定性转移可以动态地组合在一起以便简化状态机的模型图，如下例所示。

例 6.10 一个 ABRO 状态机是等待信号 *A* 与信号 *B* 的一个 FSM。当信号 *A* 与信号 *B* 都到达时，产生输出 *O*，除非重置（reset）信号 *R* 到达，这时状态机重新启动，等待信号 *A* 与信号 *B*。

该模式可以用于各种应用的建模。例如，*A* 代表一个装饰品的买家，*B* 代表卖家，*O* 代表交易发生，*R* 代表这个装饰品不能够卖出。

具体地说，该系统有 3 个布尔值输入 *A*、*B* 和 *R*，还有一个布尔值输出 *O*。当输入 *A* 与 *B* 都为 true 时输出 *O* 为 true。在任意一次迭代中，如果 *R* 的值为 true，那么系统就会回到初始状态。

图 6-18 显示了这个状态机的一种实现方法。初始状态 nAnB（“not *A* and not *B*”的缩写）表示 *A* 和 *B* 都没有到达。状态 nAB 表示 *A* 没有到达但是 *B* 到达了。

图 6-18 中的条件表达式比较复杂（尽管它可以变得更复杂，详见练习 4）。另一种实现方式更简单（但前提是你熟悉表 6-1 中总结的转移的概念），如图 6-19 所示。这个例子使用非确定性转移、默认转移和立即转移来简化条件表达式。

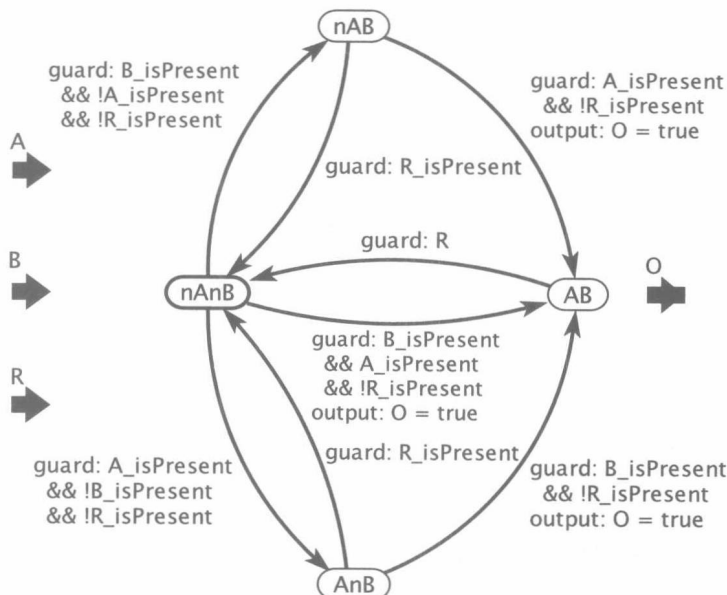


图 6-18 经典 ABRO 状态机的蛮力实现方式

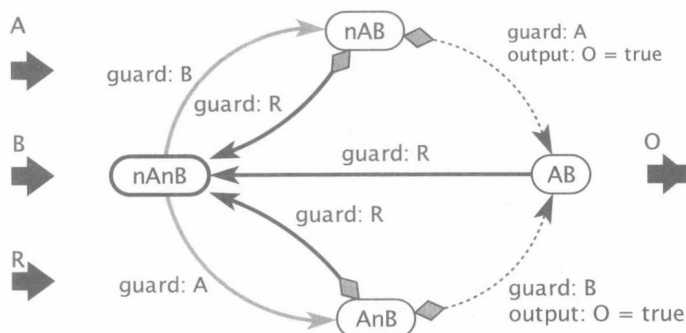


图 6-19 利用默认转移、立即转移和非确定性转移来简化条件表达式的 ABRO 状态机的实现方式

进一步探索：弱过渡态

在没有使用立即转移的情况下状态机也可能在某个状态停留的时间为 0。这样的状态称为弱过渡态 (weakly transient state)。它与 6.2.7 节中的过渡态 (transient state) 不同，不是在单一响应 (reaction) 中它有移出状态的立即转移 (immediate transition)。过渡态是一次响应的终止状态，是下一次响应的现态，但是在两次响应之间的模型时间 (model time) 为 0。注意有 (没有条件式立即转移条件的值不为 true) 默认转移的状态都是暂态，因为在进入该状态后，停留在该状态的时间只有一瞬间。

当 FSM 中的转移执行时，FSM 角色或者 ModalModel 调用上层指示器的 `fireAtCurrentTime` 方法。不管是否有其他任何附加的输入，该方法都请求下一个微步 (microstep) 中的新点火。如果指示器通过这个请求 (如计时指示器通常做的那样)，那么角色将在当前时间，一个微步后，再次点火。这保证了如果目标状态有 (在下一个微步) 立即可执行的转移，那么该转移将在模型时间变化之前执行。在模态模型 (modal model) (详见 6.3 节或者第 8 章) 中，如果目标状态有细化，那么该细化将在下一个微步的当前时间点火。这对于连续时间模型非常有用 (见第 9 章)，因为该转移可能在其他连续信号中表示不连续。这种不连续在同一个时间戳内转化成两个离散事件。

立即转移可以写入相同的输出端口，由相同迭代中发生的前一转移写入该端口。FSM 是命令式的 (imperative)，具有很好的执行顺序，所以 FSM 的输出将是转移链中写入输出端口的最终值。类似地，立即转移也可以在它们的赋值动作中给同一个参数赋值，重写前面转移的赋值动作。

6.3 分层 FSM

通过使用 ModalModel 角色而不是 FSM 角色来构建 FSM 总是可行的 (并建议这么做)。ModalModel 角色允许状态有一次或者多次细化 (refinement) 或一个或多个子模型。第 8 章将讨论这种方法的通用形式，称为模态模型 (modal model)。它的子模型可以是任意 Ptolemy II 模型。这里，只考虑子模型本身就是 FSM 的特殊情况。该方式产生了分层 FSM (hierarchical FSM) 或分层状态机 (hierarchical state machine)。

为了创建一个分层 FSM，在状态的上下文菜单中选择 Add Refinement，然后选择 State Machine Refinement，如图 6-20 所示。这就创建了一个状态机细化 (子模型)，它可以引用高层状态机的输入端口并写入高层状态机的输出端口。细化的状态本身可以继续进一步细化

(Default Refinement 或者 state Machine Refinement)。

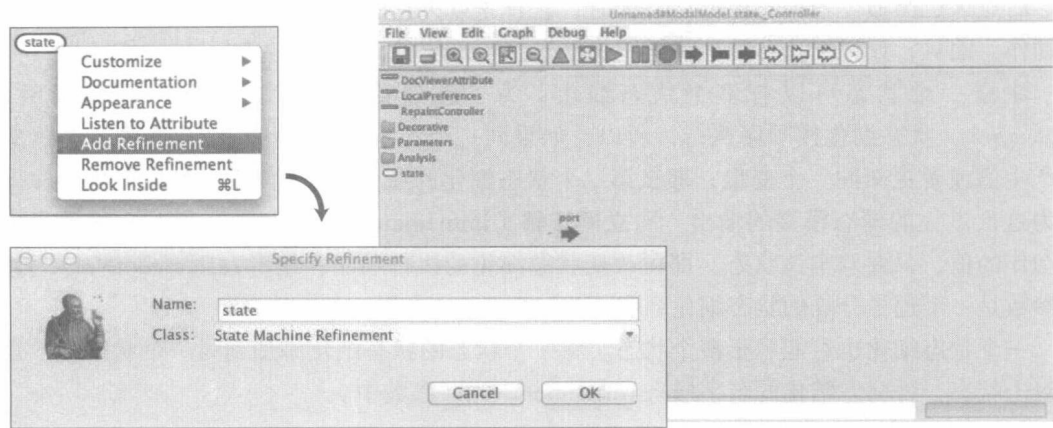


图 6-20 怎样为 ModalModel 状态添加细化

除了表 6-1 中的转移类型外，分层状态机还提供了更多的转移类型，如表 6-3 所示。具体介绍见后文。

表 6-3 分层 FSM 转移及其符号总结。这里所有的状态细化本身都是 FSM，虽然在第 8 章中可以看到任意的 Ptolemy II 模型的细化

符号	描述
	普通转移 (ordinary transition)。 在激活时，首先激活源状态的细化模型，然后，如果条件 g 的值为 $true$ (或者没有设定条件)，那么 FSM 就选择该转移。它将在输出端输出 x 被赋值为 y ，重写源状态细化在同一个端口产生的值。在转移时 (在后激活时)，角色将变量 a 赋值为 b ，再一次重写状态细化赋予 a 的值。最后，状态 $s2$ 的细化重置为初始状态。因此，这些转移有时也称为 复位转移 (reset transition)
	历史转移 (history transition)， 它与普通转移类似，唯一的不同的是，在进入 $s2$ 状态时，它的状态细化不重置为初始状态，而是从上一次细化时的状态继续执行。当第一次进入状态 $s2$ 时，细化从初始状态开始执行
	抢占式转移 (preemptive transition)， 如果现态是 $s1$ ，且条件的值为 $true$ ，那么 $s1$ 的状态细化 (FSM 的子模型) 在转移前不会被调用
	终止转移 (termination transition)， 如果状态 $s1$ 所有细化都到达了终止状态，且条件为 $true$ ，那么该转移就会发生

6.3.1 状态细化

模态模型的执行遵循简单模式。当模态模型点火时，首先，它的状态细化点火。然后计算其条件的值，并选择转移。状态细化可以产生输出，转移也可以产生输出，但是，因为状态细化首先点火，所以如果在同一个端口产生输出值，那么转移将重写由细化产生的值。执

行有着严格的顺序。

后点火也类似。当模态模型后点火时，它首先点火它的细化，然后提交并执行转移的赋值动作。而且，如果细化和转移写同一个变量，转移的赋值动作优先。

注意一个状态可以有多个状态细化。为了创建第二个状态细化，再次调用 `Add Refinement`。状态细化按顺序执行，所以，如果同一个状态的两个状态细化在同一个输出端口产生值或者更新同一个变量，那么第二个状态细化的赋值动作优先^①。最后产生的输出值成为这次点火的模态模型的输出。同**立即转移**（`immediate transition`）一样，它将重写第一个动作的值，只要双击该状态，即可改变状态细化的执行顺序，编辑 `refinementName` 参数，该参数是一个逗号分隔的状态细化列表。

一个状态细化也可能属于多个状态。将一个状态的状态细化添加到另一个状态上，只需要双击状态，将状态精化的名字插入 `refinementName` 参数中。

6.3.2 分层 FSM 的优点

下面的例子将证明分层 FSM 比扁平 FSM（`flat FSM`）更易理解、更易模块化。

例 6.11 图 6-21 显示了结合例 6.1 与例 6.8 中正常恒温器与故障恒温器的分层 FSM。

在该模型中，`Bernoulli` 角色用来产生 `fault` 信号（它的值以固定概率设置为 `true`，图中的概率为 0.01）。当 `fault` 信号为 `true` 时，这个模态模型将转移到 `faulty`（故障）状态并在返回到 `normal`（正常）模式前在这个状态上迭代 10 次。这两个状态的状态细化与图 6-12 和图 6-15 中的一样，它们分别对恒温器在正常情况和有故障情况下的行为进行了建模。

从 `normal`（正常）到 `faulty`（故障）的转移以及回到顶层 FSM，都是**抢占式转移**，它用一个起始端带有深灰色圆圈的箭头表示，这意味着当这些转移上的条件的值为 `true` 时，现态的状态细化不执行，目标状态的状态细化被重置为初始状态。相反，`faulty`（故障）状态的自循环转移是一个**历史转移**，后文将详述，当转移发生时，目标状态的状态细化因没有被初始化，所以它将从上次的断点继续执行。

图 6-22 显示了一个等价的扁平 FSM。可以说，分层图形更具可读性，能够更清楚地表示正常状态机和故障状态机以及这两个状态机之间的转移是如何发生的。本章的练习 7 更生动地说明了使用分层方法的潜在好处。

注意图 6-21 中的模型将**随机状态机**（`stochastic state machine`）与非确定性 FSM 相结合。随机状态机有随机行为，但是其采用一个明确的概率模型将通过 `Bernoulli` 角色的形式出现。非确定性 FSM 也有随机行为，但是没有概率模型可用。

6.3.3 抢占式转移与历史转移

有状态细化的状态在 Vergil 中如图 6-21 用浅灰色表示。图中顶层 FSM 使用两个新的专用转移，下面将进行其解释（详见表 6-3）。

第一个是**抢占式转移**，转移的起始端用深灰色圆圈表示。在一次点火中现态有指向另一个状态的抢占性转移，如果转移的条件为 `true`，那么状态细化没有点火。其被转移抢占

① 如果希望状态细化并发执行，可以参考第 8 章。

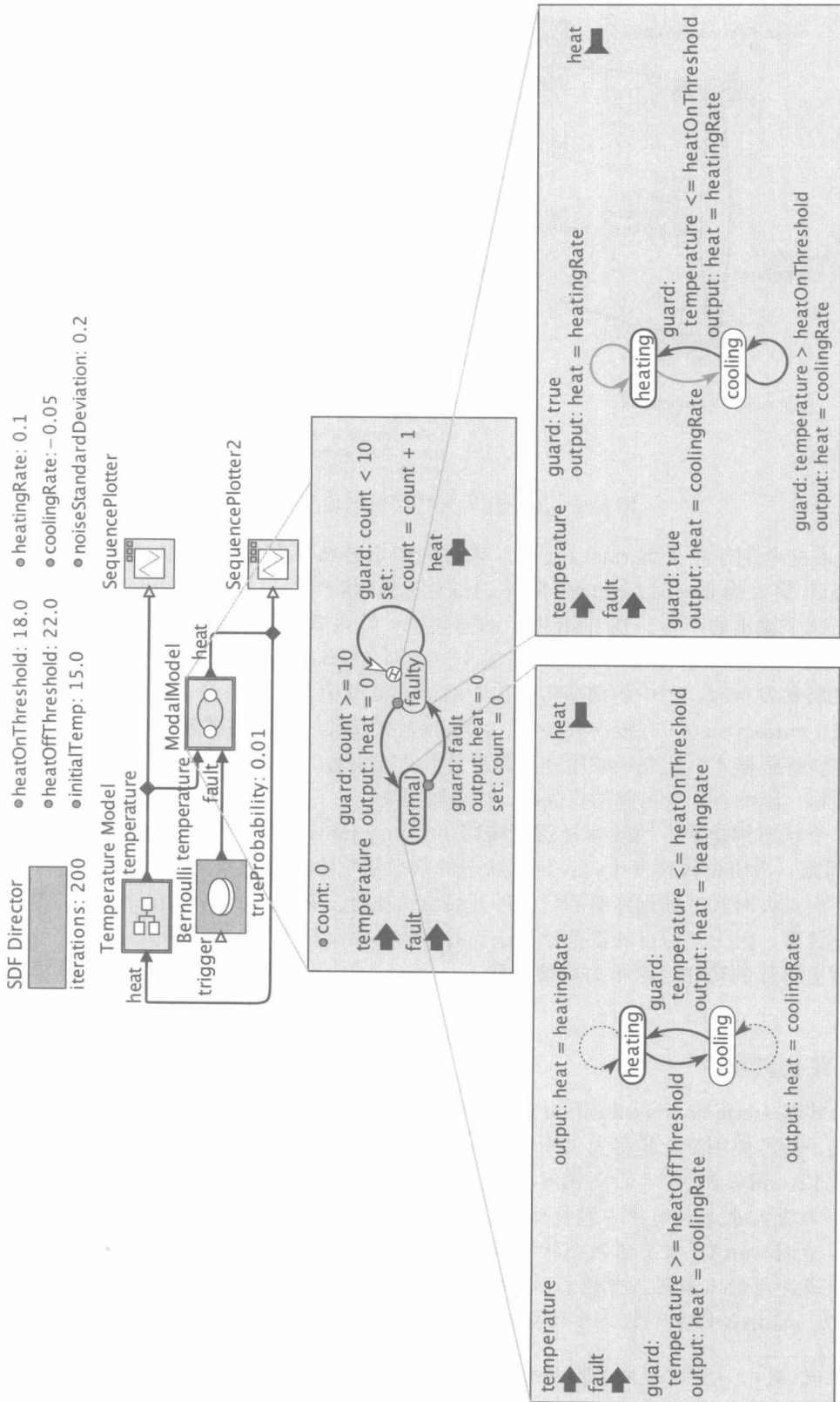


图 6-21 结合了例 6.1 与例 6.8 中正常恒温器和故障恒温器的分层 FSM

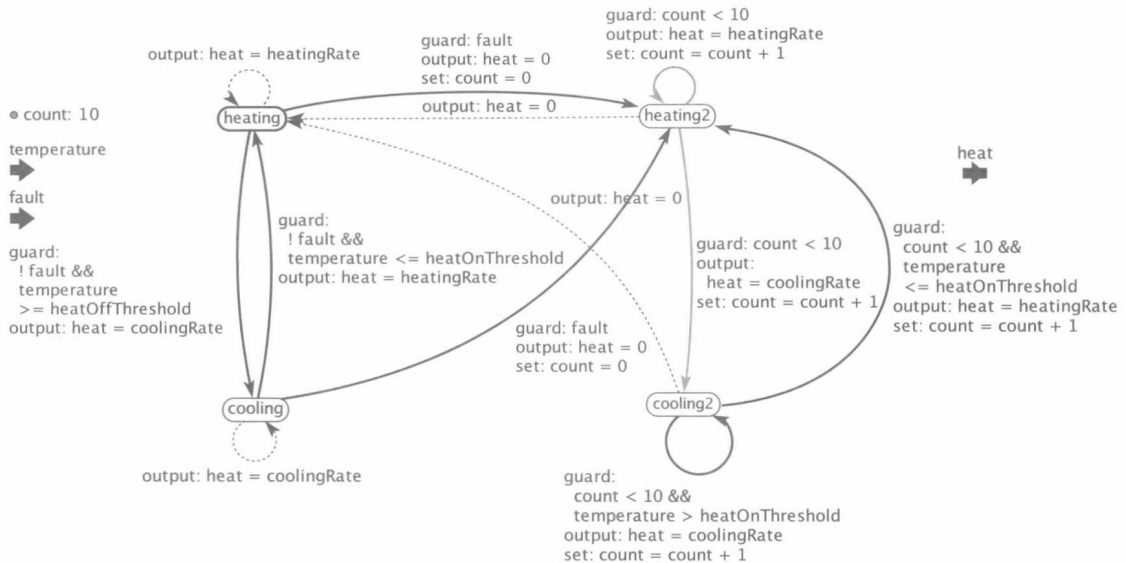


图 6-22 图 6-21 中分层 FSM 的扁平 FSM

了^①。如果这个例子中的 normal (正常) 状态的传出转移不是抢占式的, 那么在一次迭代中, 当 fault 输入为 true 时, normal 状态的细化 FSM 将产生一个正常输出。采用抢占式转移将阻止这个输出的产生。在 fault 发生的迭代中, 抢占式转移产生一个输出, 该输出不是由 normal 或者 faulty 子模型产生的正常输出。当从 normal 到 faulty 的转移或者从 faulty 到 normal 的转移发生时, 图中的模型将迭代中的 output 赋值为 0。

现态 (current state) 有抢占式转移、默认抢占式转移、非抢占式转移和默认非抢占式转移。这些转移根据 4 个优先级顺序进行条件计算。类似地, 立即转移也可以是抢占式转移或者默认转移, 因此有 4 种可能的优先级 (详见练习 9)。

第二个状态细化的专用转移是**历史转移** (history transition), 它在箭头端有一个带有字母 H 的圆圈。当历史转移发生时, 目标状态的状态细化不进行初始化, 这与普通转移不同。它从上一次点火时状态细化停留的状态处开始继续执行。在图 6-21 中, faulty 的自转移就是一个历史转移, 因为它的目的就是计算迭代的次数, 而不是干预状态细化的执行。

非历史转移的转移经常称为**复位转移** (reset transition), 因为它们重置了目标状态的状态细化。

6.3.4 终止转移

终止转移 (termination transition) 只有在现态的状态细化到达了终止状态时才可以执行。下面的例子通过使用终止转移大大简化了 ABRO 模型。

例 6.12 图 6-23 所示的是图 6-19 中 ABRO 模型的分层版本。模型的顶层是一个状态和一个由输入 R 点火的抢占式复位转移。下一层是有两个状态的状态机, 它一直在等待 waitAB, 直到 waitAB 状态的两个状态细化转移都到终止状态。该转移是终止转移, 在起始端有一个浅灰色的三角形。当终止转移发生时, 它将进入到名为 done 的终止状态, 并且产生输出 O。waitAB 的每个状态细化分别等待 A 和 B, 一旦它们接收到这两个输入, 它们就

^① 在该文献中, 这种情况有时也称为**强抢占** (strong preemption); 而**弱抢占** (weak preemption) 表示一个状态的普通传出转移, 该状态允许细化执行。

会进入终止状态。

在模态模型的每次点火中，当处在 waitAB 状态时，两个最底层的状态细化都会执行。在这种情况下，它们以哪种顺序执行并不重要。

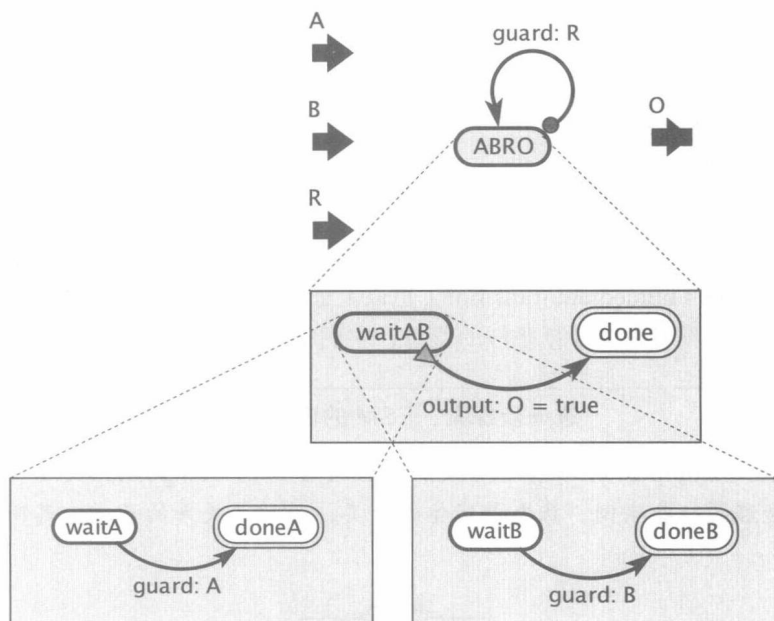


图 6-23 图 6-19 中 ABRO 模型的分层版本

分层状态机比扁平状态机更紧凑。例如，练习 5（本章的结尾）说明，如果增加 ABRO 等待的信号数量（例如，有 3 个输入的 ABCRO），那么扁平状态机的规模将迅速增大，而分层状态机只是以线性速度增大状态机规模。

通过进入状态细化中的终止状态点火的转移，有时可以当作普通转移（normal termination）使用。当子模型进入终止状态其将停止执行，且可重置子模型使其重新开始。André（1996）指出：专用的终止转移并不是必需的，因为可以使用本地信号进行代替（详见练习 6）。但是使用终止转移可以很方便地简化模型图。

6.3.5 模态模型的执行模式

ModalModel 的执行分为两个阶段，点火和后点火。在点火阶段：

- 1) 读取输入，如果有输入将输入对当前状态细化可用。
- 2) 计算当前状态的传出抢占式转移的条件值。
- 3) 如果抢占式转移可以执行，那么角色选择这个转移并执行它的输出动作。
- 4) 如果没有抢占式转移符合执行条件，那么它将：
 - a. 点火当前状态的状态细化（如果有），计算下层 FSM 转移的条件并产生所需要的输出。
 - b. 计算上层 FSM 的非抢占式转移的条件（这可能需要状态细化产生的输出）。
 - c. 执行选择的高层 FSM 转移的输出动作。

在后点火阶段：ModalModel 角色将：

- 1) 如果当前状态被点火了，那么后点火现态的状态细化，这包括执行下层 FSM 中所选

择转移的赋值动作并完成状态转移。

- 2) 执行上层 FSM 中所选择转移的赋值动作。
- 3) 将当前状态修改为所选择转移的目标状态。
- 4) 如果转移是复位转移, 那么初始化这个转移的目标状态细化。

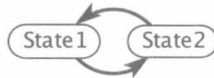
状态按以下顺序计算状态转移条件的值:

- 1) 抢占式转移。
- 2) 抢占式默认转移。
- 3) 非抢占式转移。
- 4) 非抢占式默认转移。

对于从现态 (current state) (响应的起始状态) 传出的转移, 立即转移与非立即转移并没有区别。当立即转移 (immediate transition) 也以上面的顺序 (抢占式、抢占式默认、非抢占式、非抢占式默认立即转移) 计算条件的值时, 它们唯一的区别就是目标状态。

进一步探索: FSM 的内部结构

FSM 角色是 CompositeEntity 的子类, 类似复合角色 Composite Actor。在内部, 它包含状态和转移的一些实例, 状态是实体的子类, 转移是关系的子类。其简单结构如下所示:



建模标记语言 MoML 表示如下:

```

1 <entity name="FSMActor" class="...FSMActor">
2   <entity name="State1" class="...State">
3     <property name="isInitialState" class="...Parameter"
4       value="true"/>
5   </entity>
6   <entity name="State2" class="...State"/>
7   <relation name="relation" class="...Transition"/>
8   <relation name="relation2" class="...Transition"/>
9   <link port="State1.incomingPort" relation="relation2"/>
10  <link port="State1.outgoingPort" relation="relation"/>
11  <link port="State2.incomingPort" relation="relation"/>
12  <link port="State2.outgoingPort" relation="relation2"/>
13 </entity>
  
```

相同的结构的 Java 代码为:

```

1 import ptolemy.domains.modal.kernel.FSMActor;
2 import ptolemy.domains.modal.kernel.State;
3 import ptolemy.domains.modal.kernel.Transition;
4 FSMActor actor = new FSMActor();
5 State state1 = new State(actor, "State1");
6 State state2 = new State(actor, "State2");
7 Transition relation = new Transition(actor, "relation");
8 Transition relation2 = new Transition(actor, "relation2");
9 state1.incomingPort.link(relation2);
10 state1.outgoingPort.link(relation);
11 state2.incomingPort.link(relation);
12 state2.outgoingPort.link(relation2);
  
```

因此, 从上面可以看出同一个结构的三种不同的具体语法 (concrete syntaxe)。ModalModel 包含 FSM 角色、控制器以及每个状态的状态细化。

进一步探索：分层状态机

状态机在计算理论 (Hopcroft 和 Ullman, 1979) 中有着较长的历史。分层 FSM 的早期模型是 Statecharts (状态图), 它由 Harel (1987) 提出。与 Ptolemy II 中的 FSM 相似, 状态图中的状态可以有多个状态细化; 但不同的是, 在状态图中状态细化并不是按顺序执行的。而在同步响应 (synchronous-reactive) 计算模型中, 它们大致是并发执行的。第 8 章使用模态模型 (modal model) 实现同样的效果。在 Ptolemy II 中所没有的状态图的另一个特征是, 转移可以不受层次的限制。

Esterel 同步语言也有分层状态机的语义, 虽然它给出的是文本语法而不是图形语法 (Berry and Gonthier, 1992)。Esterel 为状态机的并发复合状态提供了严格的 SR 语义 (Berry, 1999)。后来提出的 SyncCharts (同步图) 提供了一种可视化的语法 (André, 1996)。

PRET-C (Andalam et al., 2010)、Reactive C (RC) (Boussinot, 1991) 和 Synchronous C、(SC) (von Hanxleden, 2009) 都是在 Esterel 的基础上产生的支持分层状态机的语言, 这些语言都是基于 C 语言的。在 RC 和 PRET-C 中, 状态细化 (也称为“线程”) 以固定的静态顺序执行。但是, PRET-C 模型更严格, 因为不同的状态不能共享同一个状态细化。其造成的结果就是状态细化总是按相同的顺序执行。因此, Ptolemy II 的模型与 SC 的模型更接近, 它使用“优先级”的概念可以动态地决定状态细化的执行顺序。

与前面介绍的模型一样, RC 和 PRET-C 允许重复地对输出端口进行写操作, 输出端口输出的值由最后一个写入的值决定。但是, 在 RC 中, 如果重复写入操作在读操作之后发生, 系统就会抛出一个运行异常。在一次迭代中, 输出函数就像普通命令式语言中的变量一样, 本章介绍的模型与 PRET-C 中的模型更接近。与 RC 和 PRET-C 一样, 只有在迭代中最后写入的值对 FSM 的输出端口才是可见的。相反, Esterel 提供一种复合操作 (combine operator), 它将多次写入操作合并成为一个值 (例如, 通过增加几个数值)。

6.4 状态机的并发复合[⊖]

因为 FSM 可以用于 Ptolemy II 的任何域中, 并且大多数域有并发语义, 所以 Ptolemy 用户有很多方法构建并发状态机。在大多数域中, FSM 的行为与任何其他角色一样。但在某些域中, 有着细微的不同。本节主要关注的是那些在执行定点迭代 (fixed-point iteration) 的域中 (如 SR 和 Continuous 域) 构建反馈回路时出现的问题。

如之前所述, 当 FSM 执行时, 它在 fire 函数与 postfire 函数中执行的步骤不同。这种分开执行的方式对构建定点很重要, 因为在指示器寻找固定点的过程中在每次迭代中 fire 函数被调用的次数可能不止一次, 并且它不包括对任何持久状态 (persisitent state) 的改变。FSM 点火函数的 1~4 步为读取输入、计算条件的值、选择转移和产生输出——但是它们并不执行状态转移或改变本地变量的值。

例 6.13 思考图 6-24 中的例子, 它需要多次调用 fire 函数。如第 5 章所说, SR 模型的执行需要指示器在全局时钟的每个时钟节拍为每个信号寻找一个值。在第一个时钟节拍, 每

⊖ 本节在第一次阅读时可以跳过, 除非特别关注固定点域, 例如 SR 和 Continuous。

个 NonStrictDelay 角色将值写入其图标输出端口（值分别为 1 和 2）。它为 FSMActor1 定义值 in1，为 FSMActor2 定义值 in2。但是其他输入端口还没有定义。FSMActor1 的 in2 的值由 FSMActor2 提供，FSMActor2 的 in1 的值由 FSMActor1 提供。这就有可能导致因果循环，但是通过下面的讨论，它不会出现因果循环。

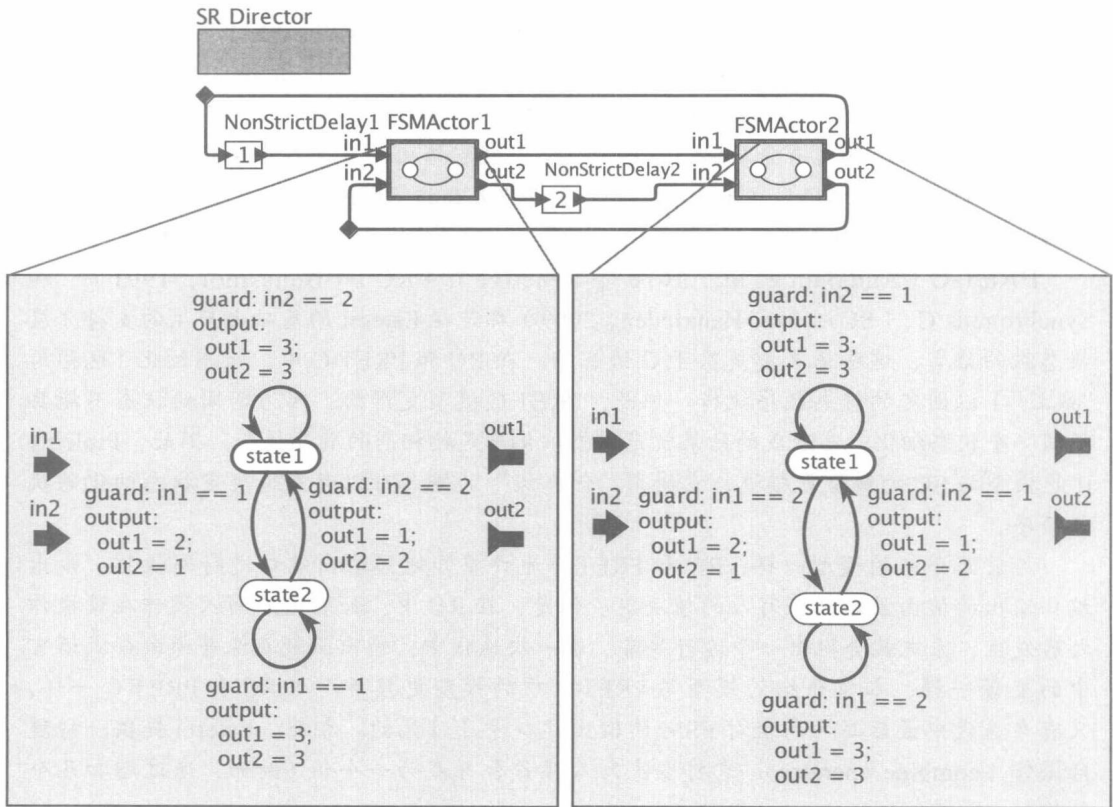


图 6-24 为了能够收敛于固定点，fire 函数与 postfire 函数之间需要分隔动作的模型

在图 6-24 中，对于 FSM 角色的所有状态，每个输入端口都有一个依赖于端口值的条件。因此，在断言任何输出前必须知道这两个输入，这里暗示了一个因果循环。然而仔细观察左边的 FSM，就可以看到从 state1（状态 1）到 state2（状态 2）的转移将在第一个时钟节拍时满足执行条件，因为 in1 有 NonStrictDelay1 产生的输入值 1。如果状态机是确定性的，那么这个转移必须是唯一满足执行条件的转移。因为在这个状态机中没有非确定性转移，假设该转移为所选择的转移。一旦做出了这样的假设，就可以宣称两个输出的值就是转移产生的值（out1 的值为 2，out2 的值为 1）。

一旦得到了这些输出值，就知道了 FSMActor2 的两个输入，并可以点火它。它的输入为 in1 = 2 和 in2 = 2，所以右边状态机从 state1 到 state2 的转移满足执行条件。该转移得出 FSMActor2 的 out2 的值为 1，所以现在知道 FSMActor1 的两个输入的值都是 1。这再次使得 FSMActor1 从 state1 到 state2 有满足条件的转移。

很容易验证，在每个时钟节拍每个状态机的两个输入有相同的值，所以每个状态都只有不超过一个的符合执行条件的传出转移。这样就保证了状态机的确定性。另外，这些输入的值总是在 1 和 2 之间随着时钟节拍而交替变换。对于 FSM Actor1，在时钟节拍 1, 2, 3...的

输入为 1, 2, 1…。对于 FSMActor2, 在时钟节拍 1, 2, 3…的输入为 1, 2, 3…。

进一步理解 6.2 节中提到的 fire 函数的 4 个执行步骤, 对理解定点迭代, 很有帮助。

1) 读取输入: 有些输入可能不是已知的。但未知的输入是不能够被读取的, 所以角色不能简单地读取这样的输入。

2) 计算现态传出转移条件的值: 有些条件可能依赖于未知的输入。这些条件可能可以计算, 也可能无法计算。例如, 如果条件表达式为 “true || in1”, 那么不管输入 in1 是否为未知, 该条件表达式都可以计算。如果条件表达式不可以计算, 那么就不对条件表达式进行计算。

3) 选择一个条件值为 true 的转移: 如果只有一个转移的条件值为 true, 那么就选择该转移。如果在同一次迭代中, 一个转移已经在前一次 fire 函数的调用中被选择, 那么角色检查这个转移当次是否被选择。如果没有被选择, 状态机将抛出一个异常并将执行暂停。不允许 FSM 在一次迭代的中途选择另一个转移。如果不止一个转移的条件值为 true, 那么角色将检查这些转移是否为非确定性转移。如果这些转移不是非确定性转移, 那么角色就抛出一个异常。如果所有的转移都是非确定性转移, 那么角色将从这些转移中选择一个。在同一次迭代中, fire 函数后续调用将选择同一个转移。

4) 如果有选择的转移, 那么执行所选择转移的输出动作: 如果选择了一个转移, 那么所有的输出都会被确定。有些输出值由转移的输出动作决定。如果没有指定它们, 那么在这个时钟节拍将它们断言为 absent。如果所有的转移都不满足条件 (即所有条件的值都为 false), 那么将所有输出设置为 absent。如果没有转移被选中, 但至少其中一个转移条件表达式的值无法计算, 那么输出是未知的。

在上述所有情况中, 如果存在满足条件的立即转移, 那么选择一个转移实际上就相当于选择一个转移链。

如前所述, 在 postfire() 函数中, 角色执行选中转移的赋值动作并从现态转移到选中转移的目标状态。一旦定点迭代已经确定了所有信号的值, 那么这些动作就会发生。如果任何信号的值在迭代结束后仍然没有定义, 那么这个模型是有缺陷的, 系统将抛出错误信息。

在执行定点迭代的域中执行的非确定性 FSM 有细微的不同。可以构建这样一个模型, 在该模型中的一个定点有两个符合执行条件的转移但转移的选择不是随机的。可能会选择那个曾经选择过的转移。这一般发生在下面这种情况下: 当在定点迭代中多次调用 fire 函数并在第一次调用时, 其中一个转移的条件由于它所需要的输入未知而无法计算。如果另一个条件在第一次调用 fire 时可以计算, 那么将总是选择另一个转移。因此, 对于非确定性状态机, 行为可能取决于定点迭代中点火的顺序。

注意, 默认转移也可以是非确定性的。但是, 默认转移只有在所有的非默认转移的条件值都为 false 时才会被选择。特别地, 如果默认转移的条件由于输入未知而无法被计算, 那么默认转移也是无法被选中的。如果所有的非默认转移的条件值都为 false, 且有多多个非确定性默认转移, 那么就随机地选择一个。

6.5 小结

本章介绍了 Ptolemy II 中使用有限状态机来定义角色行为的方法。可以通过 FSM 角色或者 ModalModel 角色来构建有限状态机, 其中 ModalModel 可以用来在 FSM 中构建分层

的状态细化，而 FSM 角色不支持这一功能。许多句法功能使得 FSM 的描述变得更紧凑，它们有操作变量（扩展的状态机）、默认转移、立即转移、抢占式转移以及分层状态机等的能力。转移有输出动作，当被一个转移选择时，在 fire 函数中执行该输出动作；转移有赋值动作，它在 postfire 函数中执行并用于改变变量的值。本章还简单地介绍了状态机的并发复合，但第 8 章将对这一主题进行更加深入的探讨，在第 8 章中将说明状态细化怎样使它们在其他域中成为并发的 Ptolemy II 模型。

练习

- 考虑例 6.1 中恒温器的一个变体。在该变体中，只有一个温度阈值，为了避免抖动，恒温器只让加热器在打开或者关闭这两个状态上各停留一个固定的时间。在初始状态，如果温度小于或等于 20°C ，那么它打开加热器并保持加热至少 30 秒。如果温度高于 20°C ，那么关闭加热器并保持关闭至少 30 秒。在两种情况下，经过 30 秒后系统回到初始状态。
 - 在 Ptolemy II 中创建一个上述的恒温器模型。可以使用 SDF 指示，并且假设它以每秒迭代一次的速度运行。
 - 恒温器可能有多少个状态？（注意！状态的个数应该包括任意局部变量可能值的个数。）
 - 在例 6.1 中的恒温器展示了称为滞后（hysteresis）的状态依赖行为。具有滞后的系统在绝对时间上是不相关的。假设输入是一个关于时间的函数 $x: \mathbb{R} \rightarrow \mathbb{R}$ （对于恒温器， t 时刻的温度就是 $x(t)$ ）。假设输入 x 产生了输出 $y: \mathbb{R} \rightarrow \mathbb{R}$ ，也是一个关于时间的函数。例如，在图 6-8 中， x 是上升信号， y 是下降信号。对于这个系统，如果输入不是 x 而是 x' ，且

$$x'(t) = x(\alpha \cdot t)$$

那么对于非负常数 α ，输出为 y' ，且

$$y'(t) = y(\alpha \cdot t)$$

输入上的时间轴的缩放引起了输出上的时间轴的缩放，所以绝对时间轴尺度是不相关的。新的恒温器模型具备这一特性吗？

- 第 5 章的练习 1 要构建一个能够区分双击和单击的模型。具体来说，定义一个角色，要求有一个名为 click 的输入端口和两个分别名为 singleClick 和 doubleClick 的输出端口。当一个 click 上的 true 输入后面有 N 个 absent 时，这个角色应该在 singleClick 上产生一个输出 true，其中 N 是角色的一个参数。如果第二个 true 输入出现在第一个 true 的 N 个时钟节拍内，那么角色应该在 doubleClick 上输出一个 true。
 - 使用扩展状态机来创建一个符合上述要求的角色。
 - 如果在输入端口 click 上的 N 个时钟节拍内有 3 个值 true，后面至少有 N 个 absent 时钟节拍，那么这个模型的行为是怎样的？
 - 讨论用简单 FSM 而不用扩展状态机的数学变量实现上述模型的可行性与优点。
- 嵌入式系统中常见的一个场景就是，系统中的一个组件 A 监测组件 B 的情况并具有报警功能。假设 B 提供传感器数据作为定时事件。组件 A 使用本地时钟提供本地定时事件的常规流。如果组件 B 没有在每个时钟间隔将传感器数据传送给组件 A ，那么系统就会出错。
 - 设计一个名为 MissDetector 的 FSM，它有两个输入端口以及两个输出端口。输入端口的名字分别是 sensor 和 clock，输出端口的名字分别为 missed 和 ok。当两个 clock 事件到达而其间没有 sensor 事件时，FSM 就会在 missed 上产生一个事件。当一个 clock 事件发生后紧接着产生第一个 sensor 事件（或者两个事件同时发生）时，那么 FSM 就会产生一个 ok 事件。
 - 设计一个名为 StatusClassifier 的 FSM。它从第一个 FSM 中获得输入，并决定组件 B 是否操作正常。具体来说，如果 FSM 在接收到了 warningThreshold missed 事件的过程中没有接收到 ok 事件，那么这个 FSM 就进入 warning 状态，其中 warningThreshold 是一个参数。另外，一旦

FSM 进入 warning 状态, 它应该一直停留在这个状态直到 normalThreshold ok 事件到达而其间没有另一个 ok 事件, normalThreshold 也是一个参数。

(c) 与设计实现的状态机相比, 评论这个练习中对行为的文字描述的准确性与清晰度。特别地, 至少找出一处文字表述不清的地方, 并解释在该处模型的含义是什么。

4. 图 6-18、图 6-19 和图 6-23 显示了使用有限状态机实现的 ABRO, 在例 6.10 中讨论。在这些实现的一次迭代中, 即使在同一次迭代中, A 和 B 都有输入, 当重置输入 R 到达时, 系统也不会产生输出 O 。

对上述模型进行一些修改, 使得在输入 R 到来时表现为弱抢占。也就是说, 只有输入 R 严格地早于输入 A 和输入 B 同时到达之前到达输入端口, R 才会影响到输出 O 的产生。特别地:

(a) 创建一个类似图 6-18 的弱抢占 ABRO, 只使用普通转移, 不采用分层技术。

(b) 创建一个类似图 6-19 的弱抢占 ABRO, 使用任何转移类型, 不使用分层技术。

(c) 创建一个类似图 6-23 的弱抢占 ABRO, 使用任何转移类型, 并使用分层技术。

5. 图 6-19 和图 6-23 分别展示了 ABRO 作为分层状态机和扁平状态机的两种实现方式。分别用分层和扁平的 ABCRO 模型, 要求系统等待 3 个输入, 分别为 A 、 B 和 C 。如果要等待的输入为 10 个, 采用哪种实现方式更适合? 为什么?

6. André (1996) 指出, 终止转移并不是必需的, 因为可以使用局部变量代替终止转移。与例 6.12 一样构建一个分层的 ABRO 模型, 但不能使用终止转移。

7. 例 6.11 中的分层 FSM 使用了复位转移, 当它进入复位转移时它初始化每个目标状态的状态细化。复位转移是抢占式转移, 当它执行时它会阻止状态细化的点火。如果这些转移不是复位转移和抢占式转移, 那么图 6-22 的扁平等价状态机将会更复杂。

(a) 构建一个等价于图 6-21 中分层 FSM 的扁平 FSM, 仅从 normal 到 faulty 的转移以及从 faulty 回到 normal 的转移为非抢占式转移。

(b) 构建一个等价于图 6-21 中分层 FSM 的扁平 FSM, 除了从 normal 到 faulty 的转移以及从 faulty 回到 normal 的转移是抢占式转移外, 如图 6-21 所示, 其他转移为历史转移而不是复位转移。

(c) 构建一个等价于图 6-21 中分层 FSM 的扁平 FSM, 仅从 normal 到 faulty 的转移以及从 faulty 回到 normal 的转移是非抢占式历史转移。

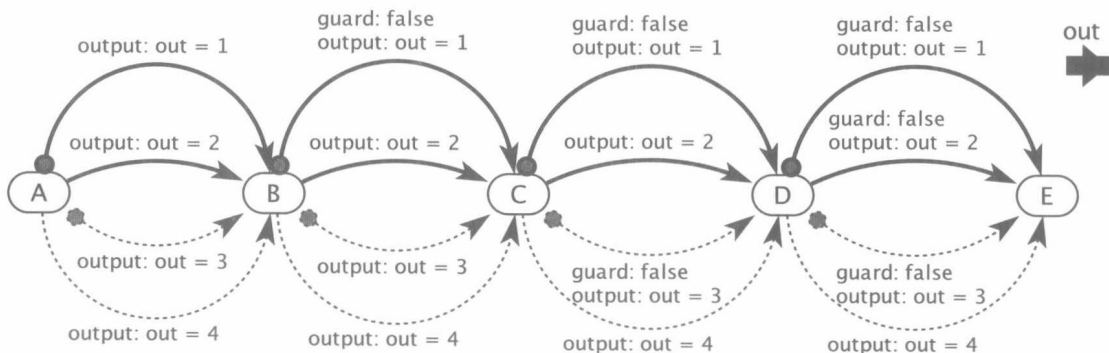
8. 思考图 6-19 中 ABRO 状态的紧凑实现。

(a) 是否可以不利用非确定性来实现一个类似的紧凑模型?

(b) 是否可以不利用非确定性来实现一个类似 ABCRO 紧凑变体?

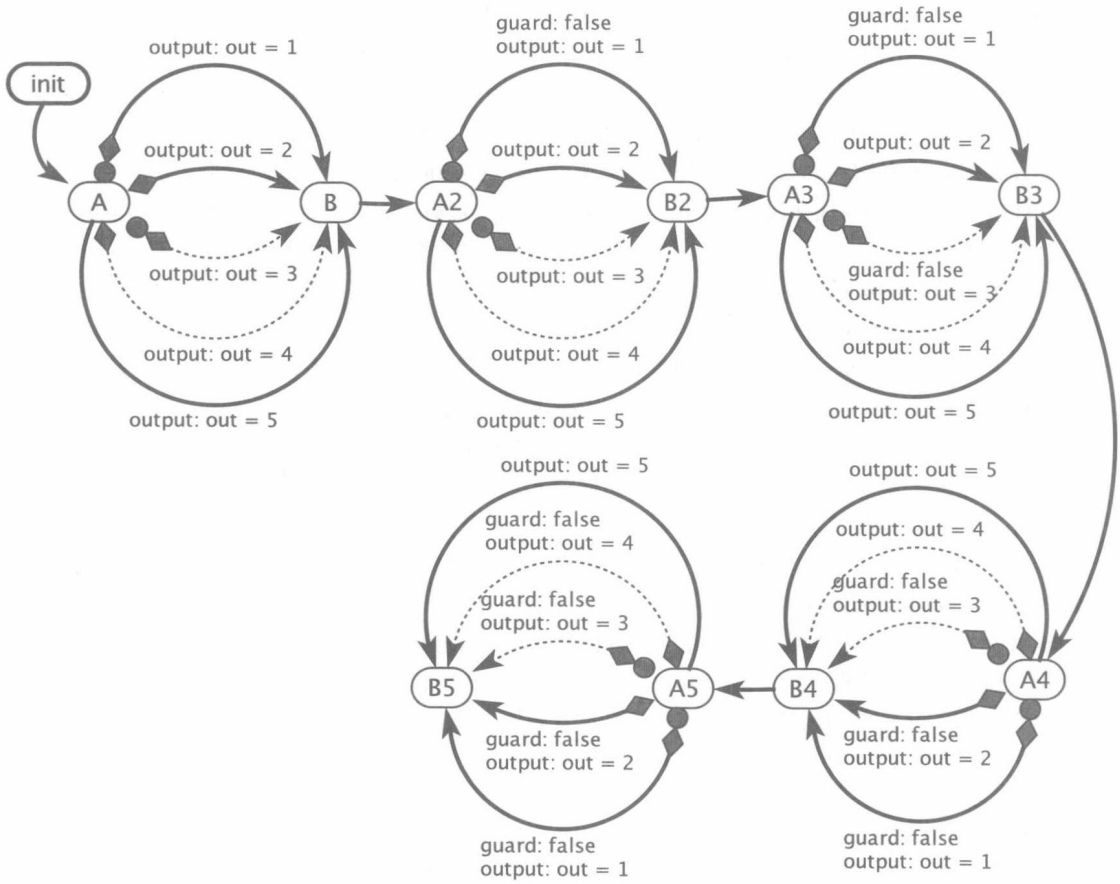
9. 该练习主要是研究转移的优先级问题。

(a) 如下的状态机:



确定前 6 个响应的输出。

(b) 如下的状态机:



请确定前 6 个响应的输出。

离散事件模型

Edward A. Lee、Jie Liu、Lukito Muliadi 和 Haiyang Zheng

离散事件（Discrete-Event, DE）域用于并发角色之间定时的、离散的交互建模。一次交互称为一个**事件**（event），事件在概念上可以理解为从一个角色发送给另一个角色的瞬时消息。在 PtolemyII 中，一个事件是指一个令牌（封装了消息的）在特定模型时间内到达端口。DE 的关键思想是每一个角色按时间顺序对输入事件做出响应。也就是说，每次角色被点火，它对输入事件的响应会迟于前点火的事件。因为这个域依赖于时间，所以它的时间模型（在本章后面讨论）对它的操作很关键。

例 7.1 图 7-1 显示了一个离散事件模型的例子。通过该例来说明 DE 的一个常见应用：对故障或错误随机发生的情况进行建模。该例对所谓的**恒定故障**建模，即在随机时间，信号停留在固定的值上，不再随输入变化。这个例子说明了事件按照时间戳顺序处理的重要性。

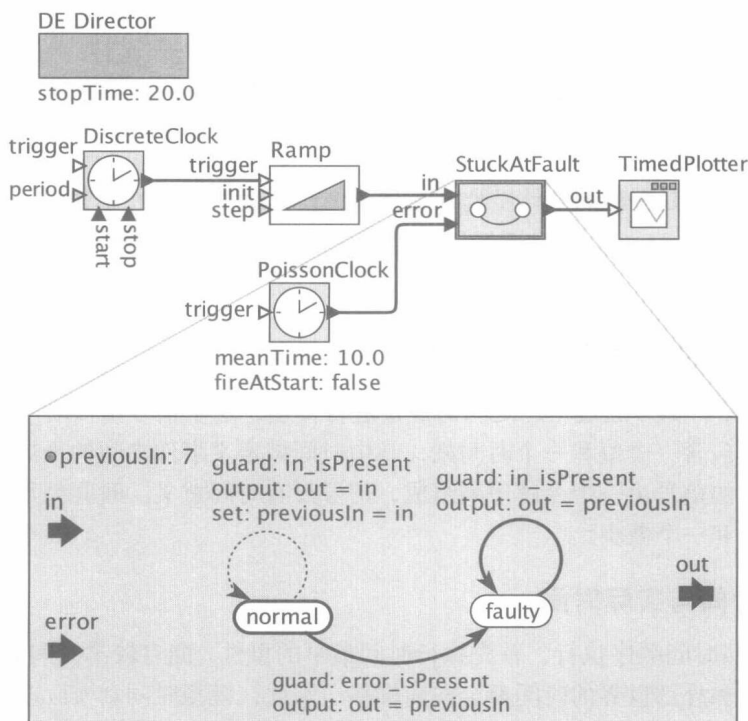


图 7-1 一个 DE 模型的简单例子

特别强调一点，StuckAtFault 角色是一个具有两种状态（normal 和 faulty）的状态机（见第 6 章）。当它处于 normal 状态时，当一个输入到达 in 端口时，输入的值就被复制到 out 端口，并存储在 previousIn 参数中。当一个 error 事件到达时，状态机就切换到 faulty 状态，从此产生一个恒定输出。

在图 7-2 中, 可以看到在时间 7~8 之间, 该模型切换到故障 (faulty) 状态。在 in 端口 (见图 7-1) 的事件由模型中的 DiscreteClock 角色激活, 它产生事件的时间间隔均匀。而激活错误条件的 PoissonClock 将产生时间间隔不均匀的事件 (代表错误的发生) (见第 7 章补充阅读: 时钟角色)。这些角色在离散事件模型中是很常见的, 重要的是输出产生的时间。而输出的值并不重要。

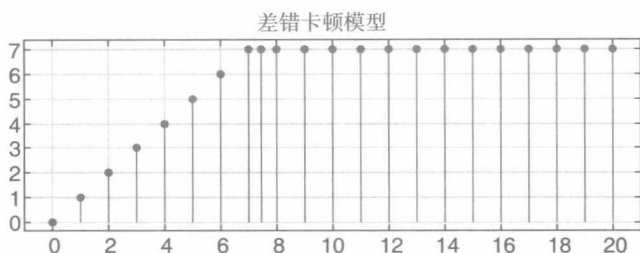


图 7-2 图 7-1 中模型的简单输出

PoissonClock 角色用来指定事件之间的预期时间间隔, 在这个模型中它的 meanTime 参数设置为 10.0, 所以这次执行发生错误的实际时间大约是 7.48, 相当接近预计时间。注意, 与 Ptolemy II 中的所有随机角色一样 (更多类似的角色见第 7 章补充阅读: 时钟角色), 可通过控制随机数生成器的 seed 来获得可重复的仿真。

一般来说, 所有通过改变状态来应对输入事件的角色, 如 StuckAtFault 角色所做的一样, 按顺序对事件做出响应是非常重要的。角色的行为取决于它的状态。这种情况下, 一旦角色有到 faulty 状态的转移, 它的输入输出行为就有很大的不同。所有后续的输入值将被忽略。(实际是一个对应关系)

本章主要探讨 DE 模型的整体结构和具体细节。从时间模型开始讨论, 然后解释 DE 如何提供一个确定性的 MoC。同时还将讨论当事件同时发生以及模型中包括反馈回路情况下的细节。

7.1 DE 域中的时间模型

在 DE 模型中, 信号通过角色之间的连接进行传递, 这个信号由按时间轴排列的事件组成。每个事件都包含一个值和一个时间戳, 其中时间戳定义事件之间的全局顺序。这与数据流不同, 数据流的信号由一个令牌序列组成, 信号没有时间意义。时间戳是超密时间值, 包含一个模型时间和一个微步^①。

7.1.1 模型时间与实际时间

DE 模型按照时间顺序执行, 首先执行时间最早的事件 (拥有较早时间戳)。随着事件的处理, 在模型内和外部世界的时间都会向前推移。因此, 模型时间和实际时间之间会产生潜在的混淆。模型时间 (model time) 是模型中的时间, 实际时间 (real time) 是模型执行时现实世界中消耗的时间 (可理解为挂钟时间 (wall-clock time))。模型时间可能比实际时间走得更快或更慢。DE 指示器有一个参数 synchronizeToRealTime, 当其设置为 true 时, 可以

① 不要混淆“模型时间”与“时间模型”的概念。在 PtolemyII 的术语中, 模型时间是一个时间值 (例如: 早上 10:15), 时间模型包含所有处理时间序列的方法。

在某种程度上让这两个时间同步。它通过延缓（如果需要）模型的执行以便实际时间“追上”模型时间。当然，它只有在计算机足够快地执行这个模型使得模型时间远远快于实际时间的情况下才起作用。当这个参数设置为 `true` 时，模型时间值以秒为单位，否则时间单位是任意的。

例 7.2 如图 7-3 中所示的 DE 模型。该模型包括一个 `PoissonClock` 角色、一个 `CurrentTime` 角色和一个 `WallClockTime` 角色（详见第 7 章补充阅读：时钟角色和知识点：时间测量）。从图中可以看出，挂钟时间和模型时间确实有很大不同；在这个例子中，执行过程中挂钟时间走得非常缓慢，而模型时间则走得很快（接近 9 秒）。因为图中横轴是模型时间，所以模型时间呈线性增加。如果把指示器的参数 `synchronizeToRealTime` 设置为 `true`，就会发现这两条线近乎完全重合。

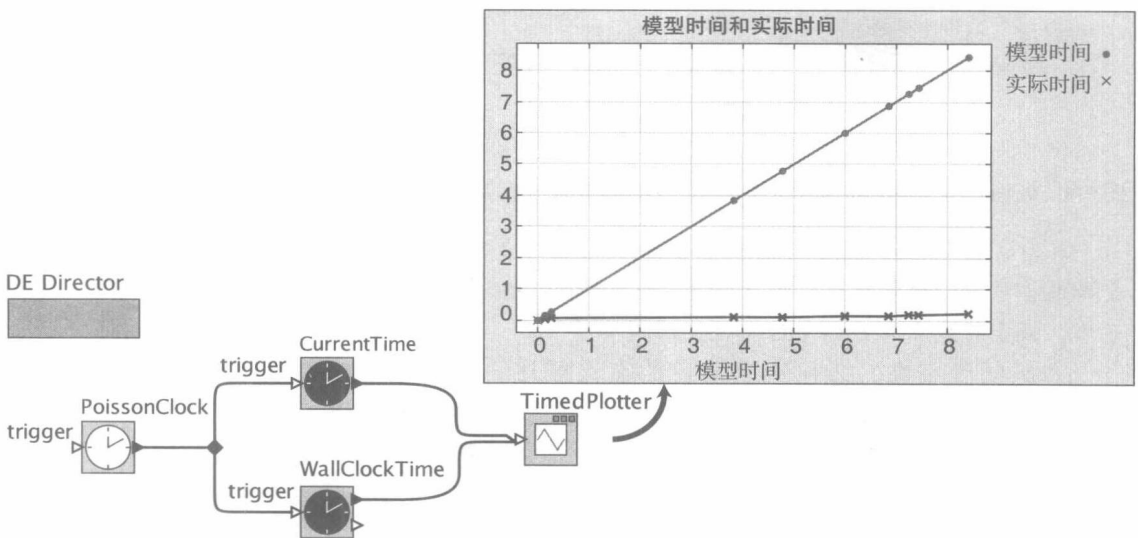


图 7-3 模型时间与实际时间（挂钟时间）

如果想准确地将模型中事件发生的时间显示给用户，将模型时间和实际时间同步是非常必要的。比如说，一个可以根据节奏产生声音的模型，如果只是尽可能快地执行它，那么该模型并不能令人满意。

7.1.2 并发事件

DE 域中存在的一个问题是：怎么处理同时事件？与前面提到的一样，强并发事件（simnltaneous event）有相同的时间戳（模型时间和微步），而弱并发事件有相同的模型时间，但是微步可以不同。已经规定了事件按照时间顺序处理，但如果两个事件有相同的时间戳，那么先处理哪一个呢？

例 7.3 考虑图 7-4 中的模型，`PoissonClock` 角色产生一个事件的时间间隔柱状图。该模型计算当前事件时间与先前事件时间的差异，从而得出如图 7-4 所示的曲线。`Previous` 角色是一个零时延角色（zero-delayactor），这意味着它产生的输出时间戳和输入时间戳相同（除了第一次点火外，因为此时不会产生输出。见第 7 章补充阅读：时间延迟）。因此，当 `PoissonClock` 角色产生一个输出时，将产生 2 个（强）并发事件，一个是在 `AddSubtract` 角色的 `plus` 端口的输入上，一个在 `Previous` 角色的输入上。

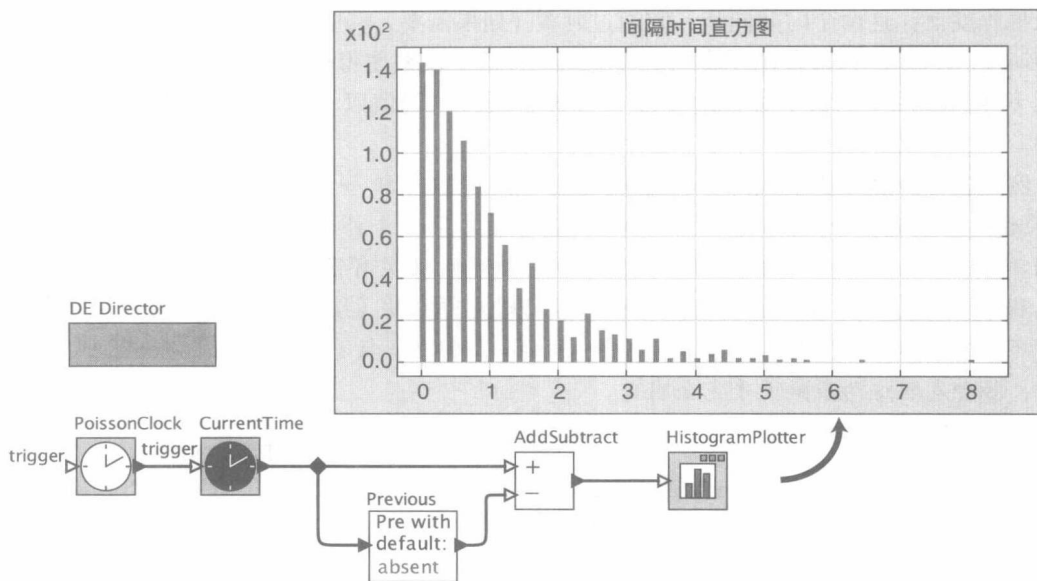


图 7-4 处理同步事件的间隔时间柱状图。这个模型有一个微小的错误，将在图 7-5 和图 7-6 中修正

此时，指示器应该先点火 AddSubtract 角色还是 Previous 角色？无论先点火哪个角色似乎都遵循事件的时间顺序，但是直觉上认为 Previous 角色应该先执行。的确如此，在这个例子中，模型将先点火 Previous 角色，原因如下。

为保证确定性，必须定义角色的点火顺序。这个顺序由模型中角色的拓扑排序 (topological sort) 来管理，它是一个按数据优先级的顺序排列的角色列表。对于一个给定的模型，也许有多种有效的拓扑排序，但是所有都必须遵循一个原则：在拓扑排序中，为角色 B 发送事件的角色 A 都会更早出现。因此，DE 指示器通过分析模型的结构，提交确定性的行为，这里产生数据的角色比使用数据的角色先被点火，即使在并发事件中也是如此。所有有效的排序产生相同的事件。

上面这个例子对理解 AddSubtract 角色如何工作很有帮助。当它点火时，它将在 plus 端口加上所有（强并发）有效令牌，在 minus 端口减去所有（强并发）有效令牌。如果在 plus 端口有令牌，但在 minus 端口没有，那么输出就是 plus 端口的那个令牌。相反，如果在 minus 端口有令牌而 plus 端口没有，那么输出就是 minus 端口令牌的负值。

基于这种行为，这里只有一种有效的拓扑排序：PoissonClock、CurrentTime、Previous、AddSubtract 和 HistogramPlotter。在这个列表中，AddSubtract 出现在 Previous 之后，因为 Previous 将把它的事件发送给 AddSubtract。因此，上例中，假定给 AddSubtract 和 Previous 输入强并发事件，指示器将总是先点火 Previous 角色。

7.1.3 同步事件

虽然图 7-4 很好地提供了一个解释角色是如何按顺序点火的例子，但是仍然有一个细节问题。AddSubtract 角色的每个输出时间都假设应该是 PoissonClock 角色的两个连续事件的间隔（间隔时间 (interarrival time)）。然而，CurrentTime 角色产生的第一个事件的值等于 PoissonClock 角色产生的第一个事件的时间戳（在这个例子中是 0.0，因为当 PoissonClock 开始执行时它默认产生一个初始事件）。可是，Previous 角色却在此时不会产生任何输出，

因为（默认）它的第一个输入产生的输出为 `absent`（见第 7 章补充阅读：时间延迟）。因此，`AddSubtract` 的第一个输出的值是 0.0，事实上，这不是一个间隔时间！所以绘制柱状图时将包括一个虚拟值 0.0。

为了解决这个问题，应该确保 `AddSubtract` 角色在每个输入端口只接收一个事件，并且只有在所有端口可接收的情况下执行接收事件。可以用 `Sampler` 角色来解决这个问题，如图 7-5 所示（详见第 7 章补充阅读：采样器和同步器）。只有当这个角色的 `trigger` 输入（下部端口）有输入事件，它才产生输出事件。一旦接收了 `trigger` 端口的输入，无论输入端口上的（强并发）事件是否可用，它都会被传送给输出。所以在该例中，`CurrentTime` 角色的第一个输出将会被丢弃，因为这时在 `trigger` 输入没有事件。

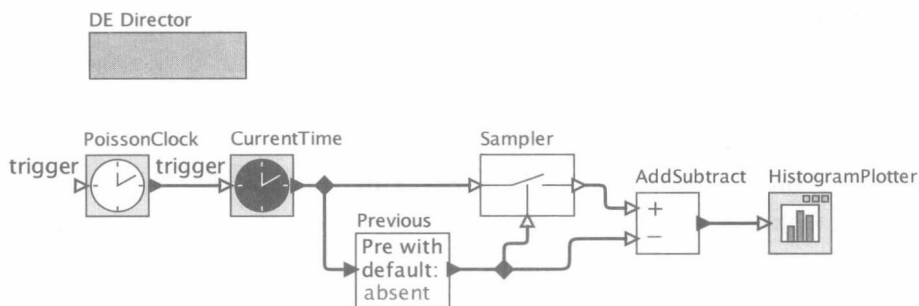


图 7-5 间隔时间的柱形图，使用 `Sampler` 角色纠正图 7-4 中的细节错误

许多其他的方法同样能够实现这个目的。比如说，第 7 章补充阅读：采样器和同步器中描述的 `Synchronizer` 角色可以替换 `Sampler`。甚至，`TimeGap` 角色也可以解决该问题（详见第 7 章补充阅读：时间测量），如图 7-6 所示。它集成了 `Precious`、`Sampler`、`AddSubtract` 角色的功能。

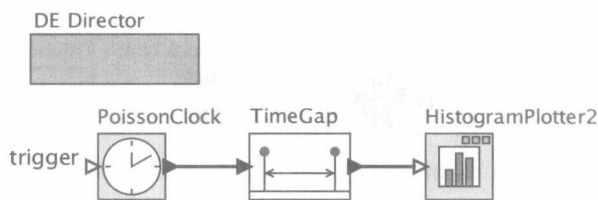
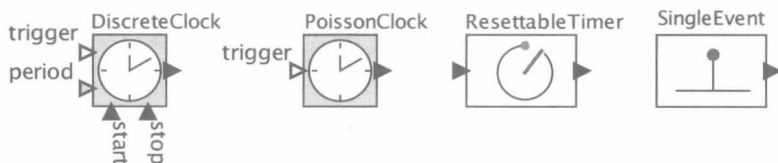


图 7-6 使用 `TimeGap` 角色的间隔时间的柱状图

补充阅读：时钟角色

时钟角色产生计时事件。以下 4 个角色都是时钟角色：



除了 `ResettableTimer` 在 `DomainSpecific` → `DiscreteEvent` 库中处，其余的都在 `Sources` → `TimedSources` 库中。

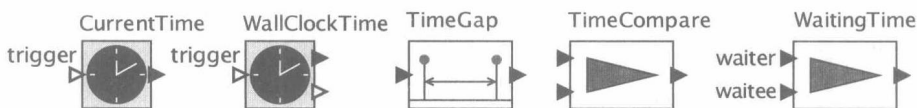
- **DiscreteClock** 是一种多用途的事件生成器。其最简单的应用是生成一系列时间间隔匀称的事件。在执行开始时（通常是时刻 0）它的默认参数使角色在一个时间单位内产生值为 1（int 型）的令牌。这种默认设置用来生成图 7-2 所示的情况。

但是 **DiscreteClock** 可以生成更复杂的事件形式，这种形式循环使用一个有限序列值并从每个周期开始时进行周期性的时间偏移。它也可以通过设置 **period** 参数为 **Infinity**（无穷大）来产生一个一次性的有限事件序列（而不是使用周期形式）。这些事件可以在任何时间内随意放置。详见角色说明文档。

- **PoissonClock**，相反，在随机时间产生事件。事件之间的时间是由独立同分布，（Independent and Identically Distributed, IID）的指数随机变量确定。一个拥有此特征的输出信号叫作泊松过程（Poisson process）。与 **DiscreteClock** 角色一样，**PoissonClock** 角色也能循环使用一个有限序列值，或者它能在各个时间产生具有相同值的事件。
- **ResettableTimer** 在输入事件指定的时间（模型时间而不是实际时间）过去后产生输出事件。也就是说，输出事件产生的时间为输入的时间与输入值之和。如果输入值是 0，那么输出将在下一个微步产生。该角色允许撤销挂起事件，也允许一个新的输入事件抢占事前预定的输出。详见角色说明文档。
- **SingleEvent** 实际并不需要该角色，因为 **DiscreteClock** 能产生单一事件。但是，该角色有时是有用的，因为它可视化地强调只有单一事件产生的模型。

补充阅读：时间测量

以下角色可以访问事件的模型时间：



CurrentTime 角色在 **Sources** → **TimedSource** 库中，用来观测事件的模型时间。当输入事件到达时它产生一个输出事件，这里输出事件的值是输入事件的模型时间（**CurrentTime** 角色的输出时间值是一个 **double** 类型，所以不能准确表示内部模型时间，1.7.3 节对此进行了解释。模型时间有一个固定精度，不会随着时间的增加而改变。相反，这个 **double** 类型的精度会随着时间增大而减小）。输出事件的时间戳和输入事件的一样，所以这个角色的响应在概念上是瞬时的。（**CurrentMicrostep** 角色在相同的库中，但上面未列出来，主要用在调试中。）

RealTime 库中的 **WallClockTime** 角色用来观测事件的实际时间。当它点火对触发事件做出反应时，其输出一个 **double** 类型值，该值代表从这个角色从初始化开始过去了多少实际时间（单位：秒）。由于这个值依赖于 DE 指示器产生的任意一种调度方法，所以该角色是非确定性的。但是，它可以用于性能测量，比方说，在模型执行的开始和结束点火它。到达 **trigger** 输入的输入事件也传递到一个输出端口，这个特征使得它较易控制某些域的下游角色调度些。

DomainSpecific → **DiscreteEvent** 库有其他用来测量事件之间模型时间差的时间相关角色。**TimeGap** 测量一个信号中连续事件之间的不同时间。**TimeCompare** 测量两个信号事件之间的时间间隔。无论事件到达哪个输入端口，都可以将其激活，并输出这个事件模型时间和另一个端口最后一个事件模型时间的时间差。**WaitingTime** 也可以测量

两个信号之间的时间间隔，但是采用的方式不同。当事件到达 waiter 端口后，该角色开始等待一个事件到达 waitee 端口。当第一个这样的事件到达后，该角色将输出模型时间差。

补充阅读：时间延迟

以下角色通过操作事件的时间戳来为延迟事件提供机制：



用处最大的是 **TimeDelay**，它通过一个指定的量来增加输入事件的模型时间。增加的时间显示在图标上，默认为 1.0。这个角色使得事件延迟（模型时间而不是实际时间）。只有模型时间被增加，输出事件和输入事件的微步是相同的。

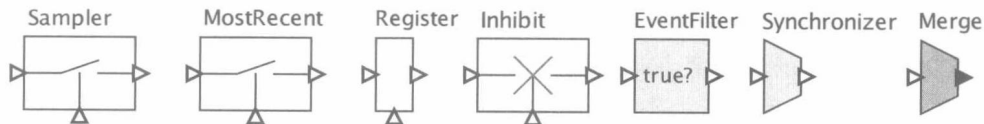
延迟量可以在下端输入端口随意给出。如果在下端输入端口提供一个事件，那么这个事件的值就会为同时或者稍后到达的输入事件指定一个延迟时间（也就是说，它们的时间戳大于或等于到达下端输入端口事件的时间戳）。**TimeDelay** 角色在 `DomainSpecific → DiscreteEvent` 库中。

有时候，设置一个模型时间延迟为 0 也很有必要。这种情况下，输出事件和输入事件的模型时间一样，但是它的微步增加了。这个设置对反馈回路很有效，在正文中有讨论。一个时延为 0.0 的 **TimeDelay** 就相当于一个 **MicrostepDelay** 角色。

根据接收的输入事件，**Previous** 角色产生一个与输入事件时间戳相同的输出事件，但其值为前一个输入事件的值。当它收到第一个事件（即前面没有事件），如果有默认值就输出默认值，否则就什么都不输出（输出为 `absent`）。因此，该角色在输入事件到达后进行延迟等待下一个输入事件的到达，再将前一个输入事件的值输出。

补充阅读：采样器和同步器

下面的角色提供同步事件机制：



Sampler（采样器）和 **Synchronizer**（同步器）在 `FlowControl → Aggregators` 库中，其余的在 `DomainSpecific → DiscreteEvent` 库中。

- 当 trigger 端口（图标的底部）接收到一个事件时，**Sampler** 就将从输入端口（多端口）选择的事件复制到它的输出端口。只有那些与 trigger 并发的输入事件被复制，这些事件将发送到相应的输出通道。
- **MostRecent** 与 **Sampler** 类似，不同的是，当它接收一个 trigger 时，将复制最新

的输入事件（这与点火事件可能并发也可能不并发）给相应的输出通道。它提供一个可选的 `initialValue` 参数，如果没有输入事件来点火就用它来指定输出值。

- **Register** 与 **MostRecent** 类似，不同的是，当它接收一个 `trigger` 时，它复制比最新事件稍微早点的事件给相应的输出通道。与 **Sampler** 和 **MostRecent** 不同，这个角色经常产生时延，因此与本章补充阅读：时间延迟中描述的延迟角色相似。延迟可以像一微步一样小，这样输入和输出就是弱并发。由于它产生延迟，所以这个角色在反馈回路中（见 7.3.2 节）非常有用。
- **Inhibit** 与 **Sampler** 相反。它复制所有的输入给输出，除非它收到一个 `trigger` 输入。
- **EventFilter** 只接收 `boolean` 型输入，并且只复制值为 `true` 的输入给输出。
- 只有每个输入通道中有一个强并发事件时 **Synchronizer** 才复制输入给输出。
- **Merge** 按照时间戳顺序将输入通道中的所有事件合并到一个信号中。如果它接收到强并发事件，那么它要么放弃除了第一个事件之外的所有的事件（如果 `discard` 参数值为 `true`），要么增加事件的微步并输出（如果 `discard` 参数值为 `false`）。

进一步探索：DE 语义

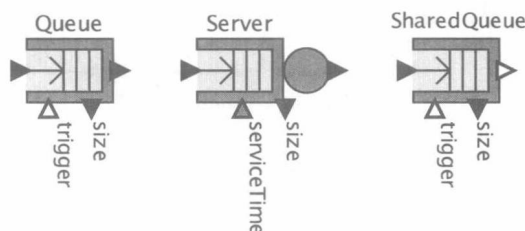
基于时间戳的离散事件模型已出现很久。最早的完整离散事件体系叫作离散事件系统规范（Discrete Event System Specification, DEVS）（Zeigler, 1976）。随着时间流逝，出现了许多差别细微的版本（比如说，Ramadge and Wonham (1989)、Cassandras (1993)、Baccelli et al. (1992) 和 Zeigler et al. (2000)）。DE 的各种变体也为各种硬件描述语言的广泛使用奠定了基础，如 VHDL、Verilog、SystemC 和许多网络仿真工具，如 OPENT Modeler（源于 OPENT Technologies 公司）、ns-2（一项 Virtual Internetwork Testbed Project，由 VINT 主导的各方合作开发的开源软件）和 ns-3（<http://www.nsnam.org/>）。在 IBM Rational 的 Rhapsody 工具 SysML 也是一个 DE 模型的变体。需要强调的是，在超密时间上，使用本章介绍的 DE 变体似乎是唯一的选择。

DE 的形式化语义是一个神奇而深邃的主题。早期方法用来描述状态机中的 DE 模型的运行（Zeigler, 1976），后来的用于定义一个度量空间（Bryant, 1985），其中角色变成了一个收缩的地图，而模型的意思变成了地图上的定点（Reed and Roscoe, 1988；Yates, 1993；Lee, 1999；Liu et al., 2006）。DE 语义也被认为是同步反应语言语义的泛化（Lee and Zheng, 2007），是基于全偏序（CPO）单调函数的一个定点（Broy, 1983；Liu and Lee, 2008）。这些后来的方法都是指称语义（Baier and Majster-Cederbaum, 1994），然而状态机方法有更多的操作风格。

DE 模型可能既庞大又复杂，因此模型的执行性能非常重要。在 DE 模型的仿真策略上仍有大量的工作需要进行。一个特别有意思的挑战是，利用硬件来并行。时间戳强调的强有序性使并行执行非常困难（Chandy and Misra, 1979；Misra, 1986；Jefferson, 1985；Fujimoto, 2000）。最近提出的一个策略叫作 PTIDES（Programming Temporally Integrated Distributed Embedded Systems），它利用网络时间同步来提供有效的分布式执行（Zhao et al., 2007；Lee et al., 2009b；Eidson et al., 2012）。在 PTIDES 中，DE 不仅是一种仿真技术，同时也是一种实现技术。也就是说，DE 的事件队列和执行引擎成为部署的嵌入式软件的一部分。

补充阅读：队列和服务器角色

下面的角色提供排队功能，它们对通信网络、制造系统、服务系统以及许多其他排队系统中的行为建模非常有用。这些角色在 DomainSpecific → Dcrefe Event 库中。



- **Queue** 在它的输入端口接收令牌并将令牌存储在队列中。当 trigger 端口接收到一个事件时，就输出队列中最前面的元素。当 trigger 端口接收到一个令牌时，如果队列中没有元素，那么就不产生输出。这种情况下，如果 persistentTrigger 参数为 true，那么接收的下一个输入将立即传送到输出端口。capacity 参数限制队列的容量；当队列满了，就丢弃接收到的输入。在每个输入处理完后，size 输出产生队列的大小。
- **Server** 对具有固定或可变服务时间的服务器进行建模。在任何给定时间，服务器要么忙（为客户端服务）要么空闲。如果当服务器空闲时一个输入到达，那么输出端就会产生一个输入令牌，它包含 serverTime 参数给定的延迟。如果当服务器忙时一个输入到达，那么输入就排队直到服务器变为空闲，此时输出端产生一个具有 serverTime 参数给定的附加延迟。如果在服务器忙时多个输入到达，那么就遵循先到先服务的原则。serverTime 可以不由参数提供而由输入端口提供。将输入端口接收的 serverTime 应用到所有同时或后来到达的事件，直到接收到另一个 serverTime。在每个输入处理完后，size 输出产生队列的大小。
- **ShareQueue** 与 Queue 类似，但它支持多路输出，每个输出都从同一队列中获得令牌。

7.2 排队系统

DE 常见的应用就是对排队系统（queueing system）进行建模，它是队列和服务器的网络。这些模型与典型的随机到达和服务时间模型配合。一种最基本的排队系统是 M/M/1 队列，这里的事件（如客户）采用泊松随机过程产生并进入队列。当事件到达队列的头部时，将被服务器在随机服务时间内进行处理。在 M/M/1 队列中，服务时间呈指数分布，如下例所示。

例 7.4 图 7-7 显示了一个 M/M/1 队列的 Ptolemy II 模型。PoissonClock 模拟客户请求到达。在该例中，平均到达间隔时间设置为 2.0。Ramp 角色用来给客户标记不同的整型值作为身份识别。图中的 ColtExponential 是 Ptolemy II 中众多随机数生成器之一（见第 7 章补充阅读：随机数生成器）。每一个客户到达时，它根据指数分布产生一个随机数。ColtExponential 角色的 lambda 参数设置为 0.5，这就使得平均值为 2.0。Server 角色是一个带有服务器（详见第 7 章补充阅读：队列和服务器角色）的队列。在该例中，每个客户到达

就指定一个新的服务时间。在图 7-7 中显示了 Server 输出的两个客户号，还显示了一个客户到达或离开时队列的大小。注意在时间点为 4 时客户突然增多，导致队列大小增大。

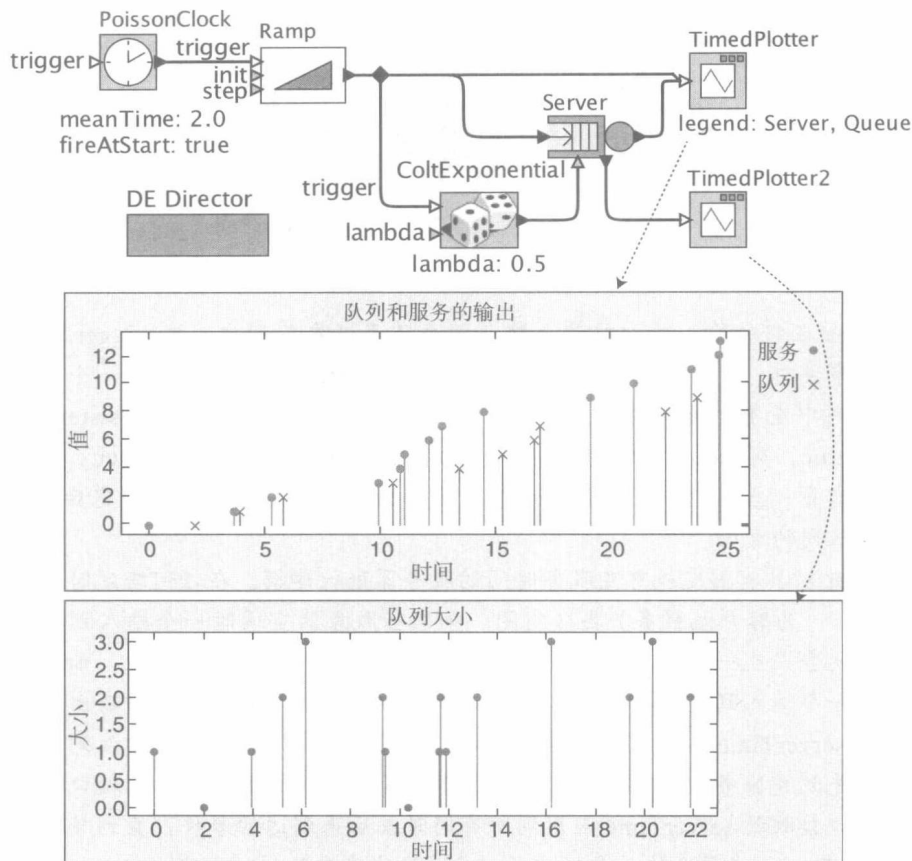
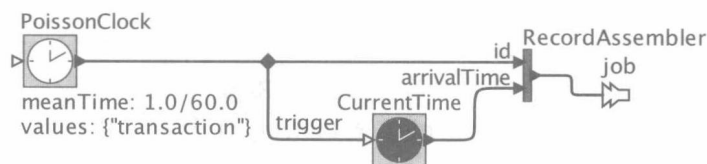


图 7-7 一个 M/M/1 队列模型

接下来将介绍更多有趣的例子使用队列和服务网络。

例 7.5 图 7-8 显示了一个存储系统模型，这里任务随机到达并由 CPU 执行。然后 CPU 写第一个磁盘，执行额外处理，写第二个磁盘，最终进行第三轮处理。这个特别的模型由 Simitci (2003) 提出，使用排队理论分析模型，预测通过网络的平均延迟时间为 0.057 秒。实验结果表明在 MonitorValue 角色显示的实验结果与针对 Monte Carlo 运行的预期值非常接近。

为了更好地测量任务的等待时延，任务以到达的时间作为时间戳。EventGenerator 复合角色实现如下所示：



该复合角色采用泊松随机过程产生事件，到达间隔的平均时间为 1/60 秒。每个事件都用两个字段 (id 和 arrivalTime) 来记录 (详见第 7 章补充阅读：记录角色)。arrivalTime 带

有 CurrentTime 角色产生的事件的时间戳。id 为一个字符串常量“transaction”，它用于将事件路由到合适的磁盘。图 7-8 中，CPU、Disk1 和 Disk2 角色也是复合角色，每个都是面向角色类 ExponentialServer 的实例。定义如下：

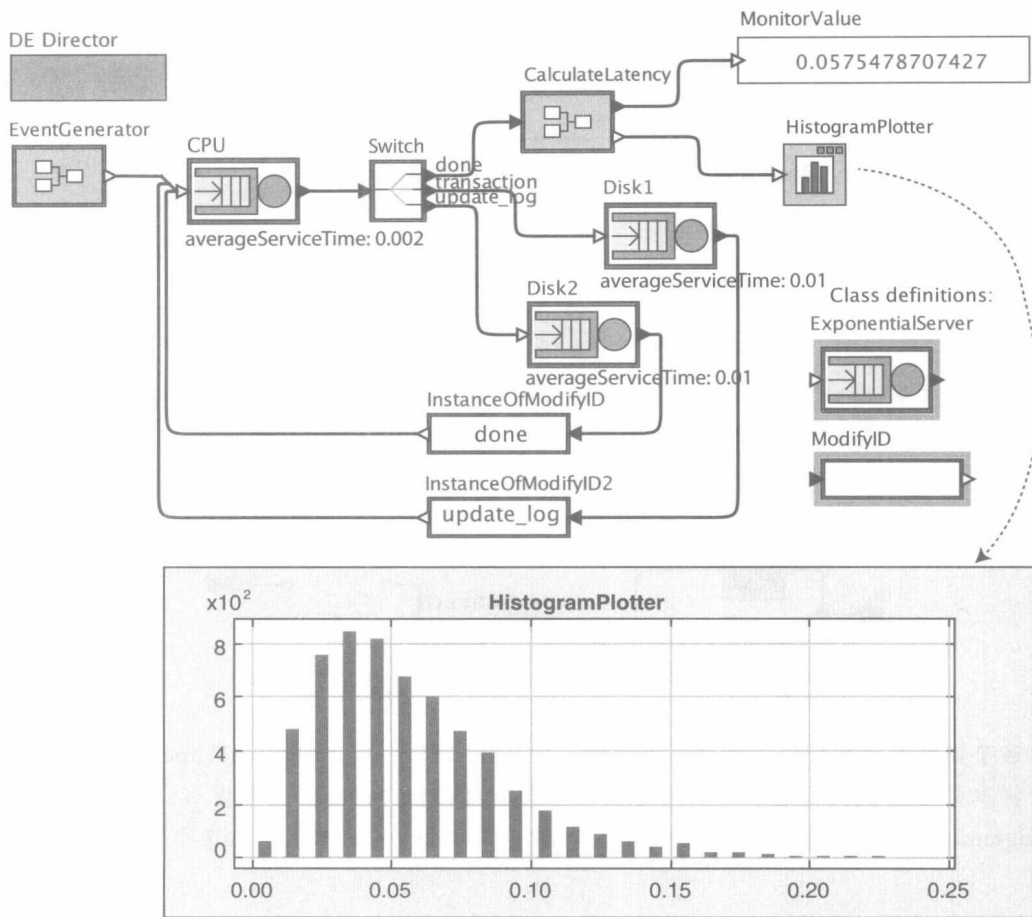
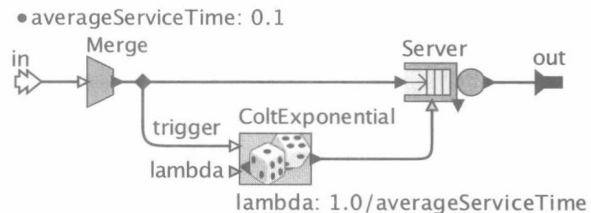


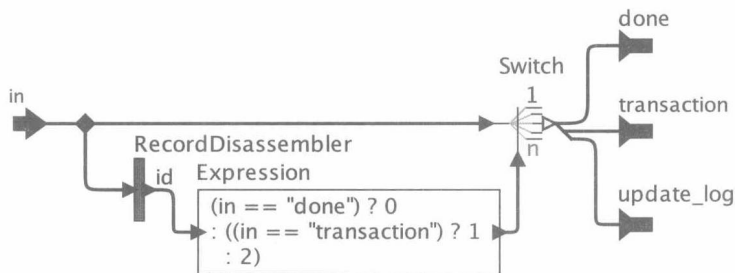
图 7-8 一个写两个磁盘的事务处理排队模型



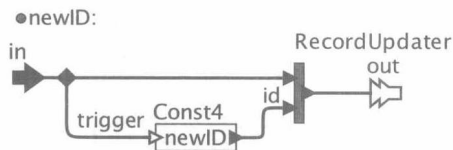
该复合角色只有一个参数 averageServiceTime。它实现了一个带有随机服务时间的服务器。一旦事件到达输入通道，ColtExponential 角色就根据指数分布生成一个随机数。随机数指定新到达事件的服务时间。如果队列为空，事件将马上执行。否则当队列中该事件前面的事件执行完后才执行它。

再次参考图 7-8，Switch 定义如下：

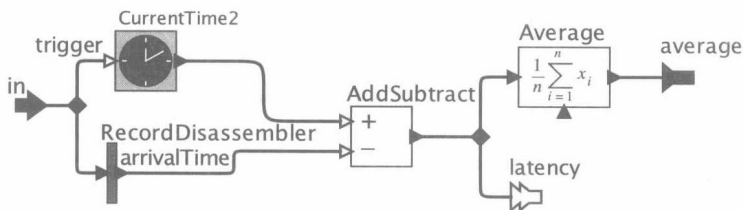
这个角色从输入记录中提取 id 字段，并使用 Expression 和 Switch 角色，根据 id 将输入事件路由到 3 个输出端口之一。



包含面向角色类 **ModifyID** 实例的两个反馈路径定义如下：



通过使用 **RecordUpdater**（详见第 7 章补充阅读：记录角色）替换记录的 id 字段，该角色从输入记录构建一个新的记录。拥有新 id 的事件再次被 CPU 处理。当 id 变成了“done”时，这个任务事件就被路由到 **CalculateLatency** 复合角色，该角色的实现如下：



该子模型负责测量任务的总延迟时间。它从输入记录中提取 **arrivalTime**，并从已完成任务中提取的时间戳减去它。然后同时输出计算的延迟和 **Average** 角色得出的平均执行延迟。**HistogramPlotter** 角色用来绘制延迟的柱状图，**MonitorValue** 用来显示这个模型运行的平均时间。

7.3 调度

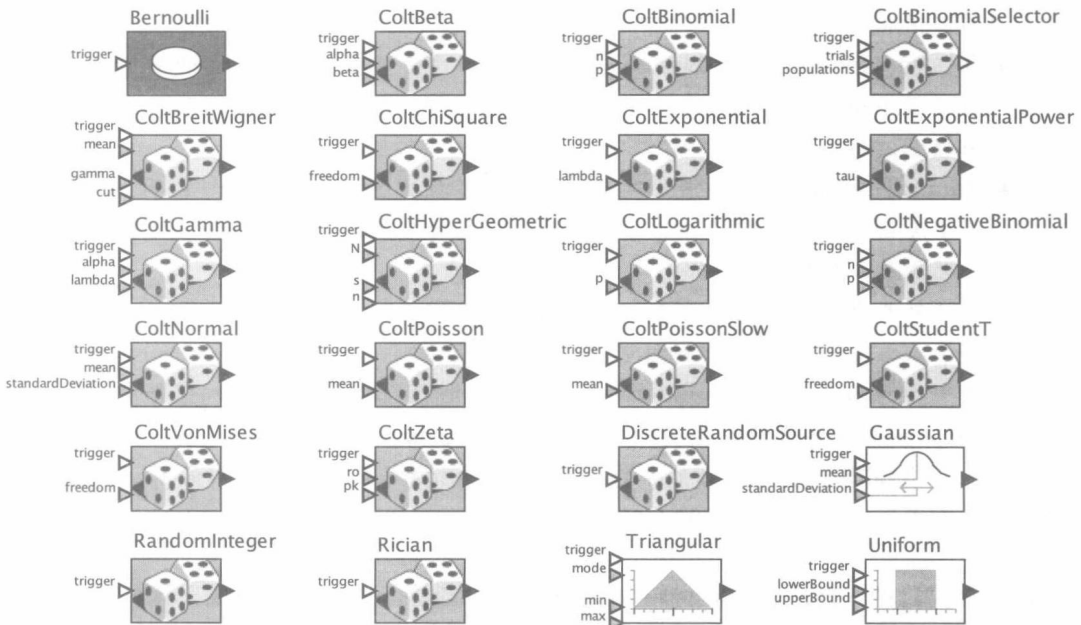
DE 的主要任务是根据时间戳顺序点火角色。DE 指示器维护一个事件队列，该队列按时间戳排序，排序时先考虑模型时间，在模型时间相同的情况下再按微步排序。DE 指示器选择事件队列中最早的事件开始执行，并把事件的时间戳改为当前模型时间（current model time）。然后它确定分配给该事件的角色，并在事件队列中寻找指定给相同角色的具有相同时间戳的所有其他事件。然后把这些事件发送到该角色的输入端口，点火角色。如果角色点火后没有执行完所有的事件，那么指示器再次点火该角色直到所有输入事件都执行完毕^①。

DE 指示器使用特殊规则来保证确定性。特别是，它保证在点火角色前，给所有事件提供的时间戳等于当前时间戳。随后它不能具有相同时间戳的附加事件。该附加事件也许有相同的模型时间，但是微步不相同。

① 注意，如果一个角色被写错并且没有执行输入事件，那么这个策略将导致无限循环，这个角色就会不停地点火。这说明角色本身存在错误。在 DE 域中，希望角色通过读输入来消除这个错误。

补充阅读：随机数生成器

Ptolemy II 中有很多随机数生成器，如下所示：



每次被点火时这些角色都在其输出端口产生一个随机数。大部分角色都有在输入端口设置的参数，所以每次点火时这些参数都可能不同（详见角色说明文档）。

以上角色都有一个 seed 参数，可用于确保可重复的随机序列。当你设置一个随机角色的 seed 参数时，所有角色的 seed 参数都被设置了（这是一个共享参数，意味着该值被模型中拥有这个参数的所有角色共享）。所有随机角色还共享一个基础随机数生成器，所以很容易控制进行重复试验。

前缀为 Colt 的角色由 David Bauer 和 Kostas Oikonomou 设计，他们借鉴了其他已有成果，使用一个最初由 CERN（欧洲核子研究组织）开发的一个名为 Colt 的开源库。

补充阅读：记录角色

Ptolemy II 中的记录类型类似于 C 语言中的 struct（结构体）。它可以包含任意个命名的字段，每个字段具有任意数据类型（甚至是一个记录）。在菜单 [Actors → FlowControl → Aggregators] 中有许多角色可以操作记录，如下所示。



所有这些角色需要在使用角色的 [Configure → Ports] 菜单项添加新端口。建议使用 Show Name 功能将字段名清晰地显示出来，如图 2-18 所示。

- RecordAssembler 输出一个记录，其中包含每一个输入端口的一个字段，这个字

段名和输入端口名一样。

- **RecordDisassembler** 从记录中提出字段。输出端口名必须与字段名相符。如果输入记录没有与输出端口名相符的字段，那么类型系统（见第 14 章）将报告一个类型错误。
- **OrderedRecordAssembler** 构建一个记录令牌，其中记录中字段的顺序与输入端口的顺序相同（自顶向下）。这个角色一般不用，除非你要写遍历该记录的字段并依赖于该顺序的 Java 代码。
- **RecordUpdater** 增加或者修改一个记录的字段。图标的 built-in（内置）输入端口提供原始记录。必需增加额外的输入端口并命名为想要增加或修改的字段名。输出记录是一个修改记录。

还有另外两个对构造记录非常有用的角色，可以在



[Actors → Conversions] 菜单中找到它们，如右图所示：

这两个角色都可以接收字符串输入。**ExpressionToToken** 解析输入字符串，它可以是第 13 章介绍的表达式语言接受的任意字符串，包括记录。如果输入字符串指定了一个记录，那么输出令牌就是记录令牌。**JSONToRecord** 接受广泛使用的 Internet JSON 格式的任意字符串，并生成一个记录。

同前面讨论的一样，DE 指示器对角色进行拓扑排序。一旦排序完成，就给每个角色分配一个等级（level）。这个等级是从源角色（源角色没有上游角色）或延迟角色沿着路径历经的最大上游角色的个数。

例 7.6 例如，图 7-5 所示的角色有下面的等级：

- PoissonClock: 0
- CurrentTime: 1
- Previous: 2
- Sampler: 3
- AddSubtract: 4
- HistogramPlotter: 5

当具有相同时间戳的两个事件插入事件队列中时，具有较低等级角色的事件将排在队列的前面。例如，在图 7-5 中，这就保证了 Previous 先于 Sampler 点火，Sampler 先于 AddSubtract 点火。因此，在 AddSubtract 点火时，可以确保所有的输入事件都处在当前模型时间内。

7.3.1 优先级

DE 指示器根据模型时间，然后根据微步，最后根据等级对事件排序。但是也有可能出现具有相同时间戳和等级的事件。

例 7.7 仔细观察图 7-9 中的模型。这里，两个 Ramp 角色的等级为 1，两个 FileWriter 的等级为 2。当时钟产生一个事件时，事件队列中将有两个具有相同时间戳和相同等级的事件，一个分配给 Ramp，另一个给 Ramp2。因为这两个角色不会进行通信，所以它们点火的顺序无关紧要。然而，两个 FileWriter 角色可能写相同的文件（或进行标准输出）。这样，点火顺序就十分重要，但时间戳和等级不能单独决定点火顺序。不希望指示器随意选择一个顺

序。在这种情况下，设计者可以选择使用优先级参数，如下所述。

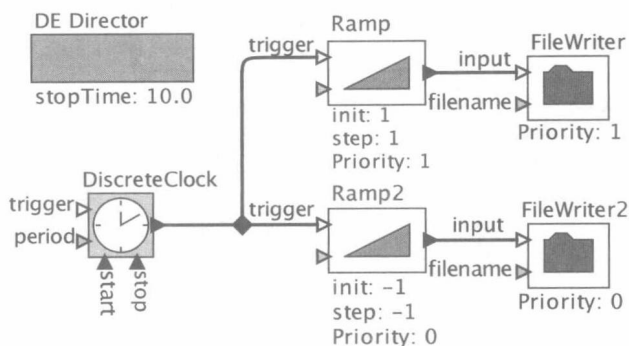


图 7-9 尽管优先级很少使用，但是在 DE 中设置角色的优先级有时却十分有用。它只有在点火顺序既不能由时间戳决定也不由数据优先级决定时有用

前面的例子展示了一个不同寻常的情况：在模型中，两个角色即便彼此没有直接通信，还是会相互影响。它们以一种 DE 指示器不可见的方式偷偷地（under the table）相互作用。当发生这种相互作用时，模型设计者希望能控制执行的顺序。

在 Utilities → Parameters 库中有一个 Priority 参数可以拖曳到角色上。可通过双击角色为其独立赋值，较低的值代表较高的优先级。当事件有相同的时间戳和相同的等级时，DE 指示器查询目标角色的优先级，并把优先级高的（Priority 值低）角色的事件放在事件队列的前面。

例 7.8 对图 7-9 中的例子，Ramp2 和 FileWriter2 的 Priority 值为 0，所以它们将先于 Priority 值为 1 的 Ramp 和 FileWriter 点火。如果不使用优先级，顺序将无法确定。

在 DE 模型中，很少使用优先级，因为它们的值是全局性的（也就是说，在模型中只有特殊情况才使用优先级），这种机制不适合模块化操作。

7.3.2 反馈回路

如果模型有一个有向回路（称为反馈回路，(feedback loop)），那么就不可能在 DE 域中进行拓扑排序。每个反馈回路都要求至少有一个产生时间延迟的角色，如 TimeDelay、Register 或者队列，或者服务器（详见第 7 章补充阅读：时间延迟、知识点、采样器和同步器；补充阅读：队列和服务器角色）。

例 7.9 仔细观察图 7-10 中的模型。该模型有一个 DiscreteClock 角色，它每 1.0 个时间单位产生事件。这些事件点火 Ramp 角色，它从 0 开始产生输出事件，每次点火都增加一个事件。在该模型中，Ramp 的输出传送到 AddSubtract 角色，这个输出经历了一个时间单位的延迟，从 Ramp 的输出中减去它本身先前的输出，结果在图表中显示。

为了保证反馈模型是确定性的，DE 指示器将延迟角色的等级设为 0，但是将它们读取输入的时间延迟到后点火阶段，这将在同时运行的其他角色点火之后发生。

例 7.10 在图 7-10 中，TimeDelay 的等级为 0，而 AddSubtract 的等级为 1，所以当计划产生输出时，TimeDelay 在任何时间戳都在 AddSubtract 前点火。但是，TimeDelay 不会读取输入的内容，直到 AddSubtract 角色已经点火以便响应它自己的输出事件。TimeDelay 在后点火阶段读输入，在这个阶段它简单地记录输入并请求一个新的点火，该点火发生在当前时间加上它的时间延迟（这里为 0.1）。

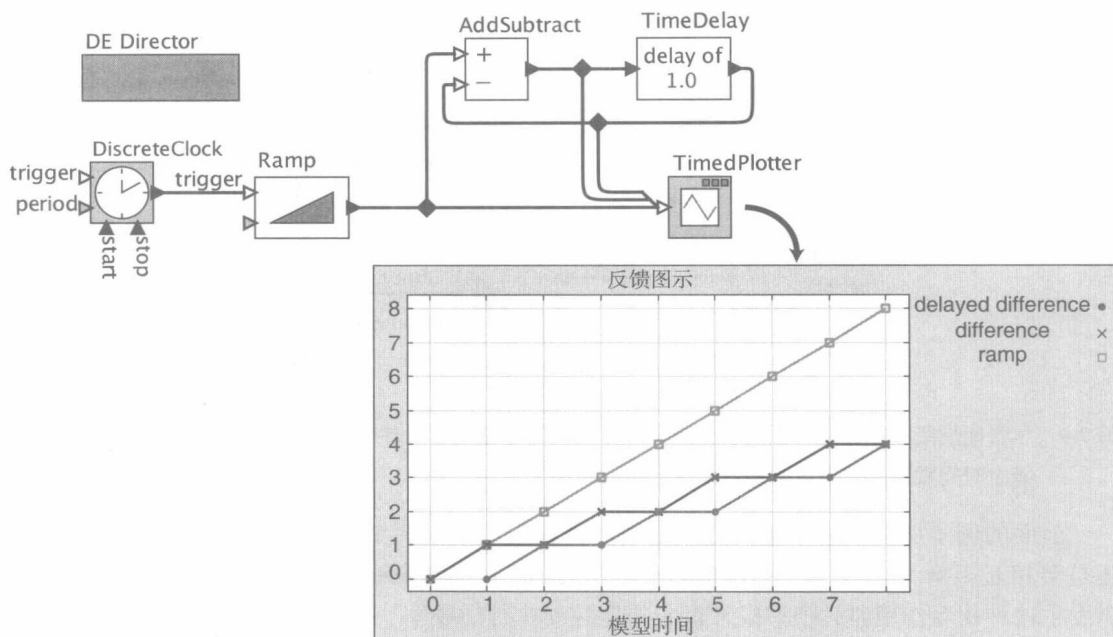


图 7-10 具有反馈的离散事件模型，它需要一个类似 TimeDelay 的延迟角色

有时候，有必要在反馈回路中放置一个延迟为 0.0 的 TimeDelay 角色。这样能起到增加微步而不增加模型时间的效果，以允许没有时间推进的迭代。

例 7.11 观察图 7-11 中的模型，它产生的曲线如图 7-12 所示。该模型使用的反馈回路在每个整数模型时间产生数目不定的事件，所使用的反馈回路有一个延迟设置为 0.0 的 TimeDelay 角色。这使得反馈的事件在相同模型时间内使用递增的微步。这个模型使用 BooleanSwitch (见第 3 章补充阅读：令牌流控制角色) 反馈一个非负值的令牌，只要其值为非负的。

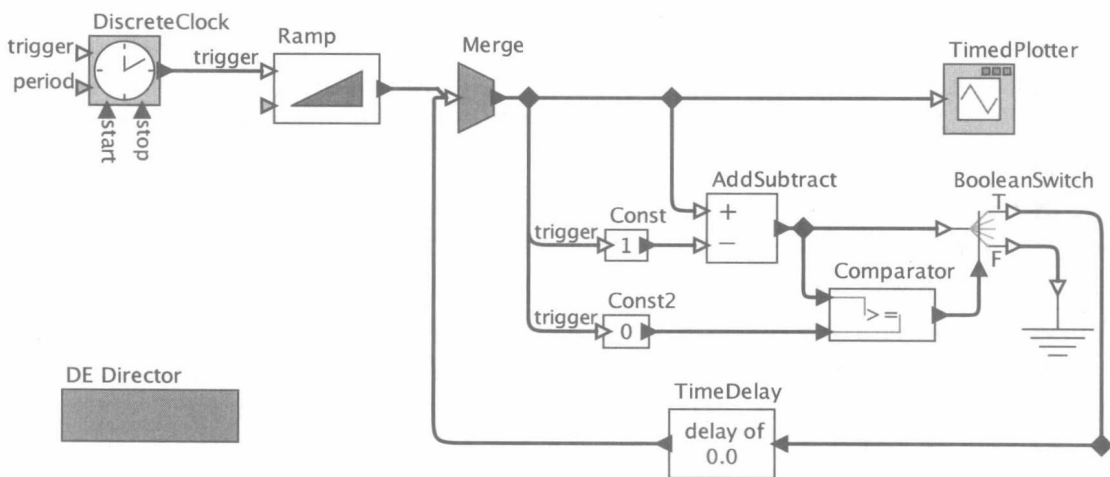


图 7-11 延迟值为 0 的 TimeDelay 的说明

前面这个例子说明了在 DE 中的迭代，它使用一个类似于例 3.11 说明的数据流中反馈迭

代的结构。在 DE 中, 使用 Merger 角色而不是 BooleanSelect 角色, 因为时间戳的使用使得 Merge 是确定性的。

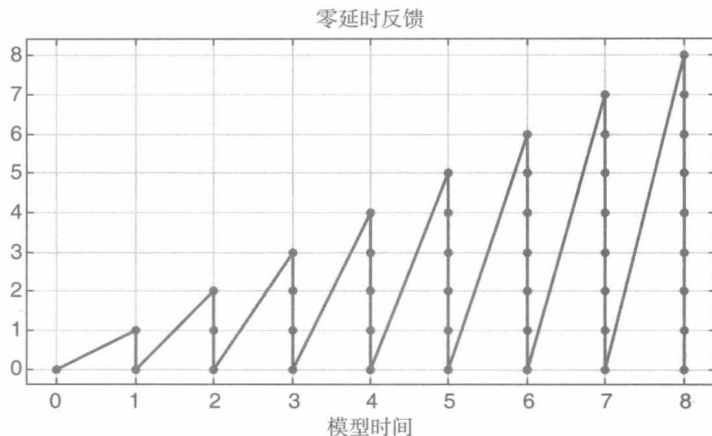


图 7-12 执行图 7-11 中模型的结果

7.3.3 多线程执行

DE 指示器一次点火一个角色, 直到前一个点火结束才能点火下一个角色。这种方法带来了两个潜在的问题。第一, 如果一个角色没有从其 fire 函数返回, 那么整个模型都被阻塞。如果角色企图执行 I/O, 那么这个问题可能出现。第二, 这种模型的执行不能利用多核体系结构。使用 HigherOrderActors 库中的就有 ThreadedComposite 角色可以解决上述两个问题。下面的例子阐述第一个问题。

例 7.12 观察图 7-13 中的例子, 它使用在 4.1.1 节中介绍的 InteractiveShell 角色。在这个模型中, Expression 角色用于格式化一个需要显示的字符串 (详见 13.2.4 节)。

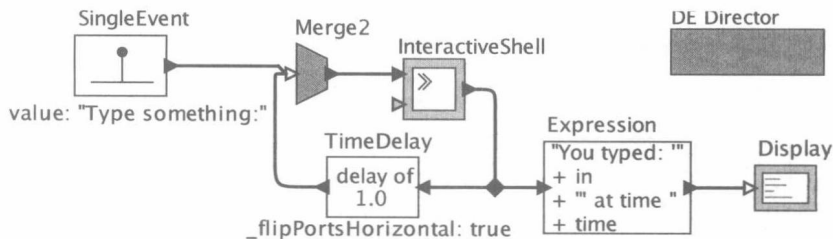


图 7-13 一个使用 InteractiveShell 角色的 DE 模型, 只有接收到用户的输入时, 该角色才能点火执行

注意, 这个模型中的时间戳并没有特别的意义。它们不能准确地反映用户输入一个值的时间, 但它们能够表示用户输入的顺序。即输入越晚时间戳的值越大。此外, 在 InteractiveShell 角色等待用户输入时, 该模型不能做任何事情。

ThreadedComposite 的角色是一个高阶组件的例子, 它使用另一个角色作为参数或者输入。ThreadedComposite 参数通过另一个角色来设置, 当它点火时, 它本身不执行任何操作, 仅执行另一个线程中的其他角色, 并马上从 fire 函数中立即返回。由于角色的功能在另一个线程中执行, 所以该模型就没有阻塞。更有趣的是, 在保证结果确定性的同时, ThreadedComposite 可以并发执行。因此, 这个功能可以用来创建一个更有用的交互模型, 如图 7-13 所示范例, 运行结果见图 7-14。

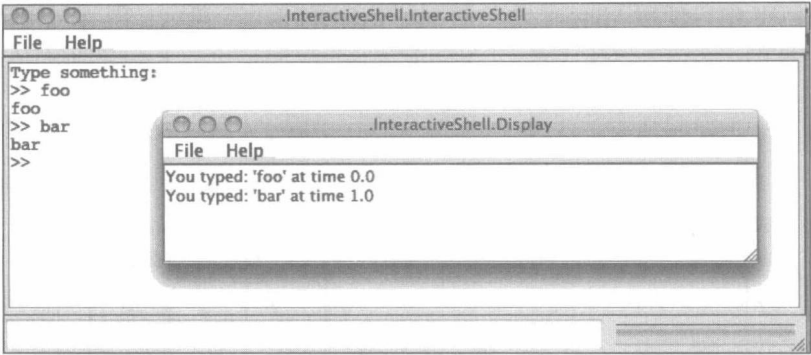


图 7-14 图 7-13 中模型的执行结果

例 7.13 仔细观察图 7-15 中的模型。该模型打开了两个交互式命令输入窗口，如图 7-16 所示。如果用户没有输入，该模型也不会阻塞，因为其采用 ThreadedComposite 角色。

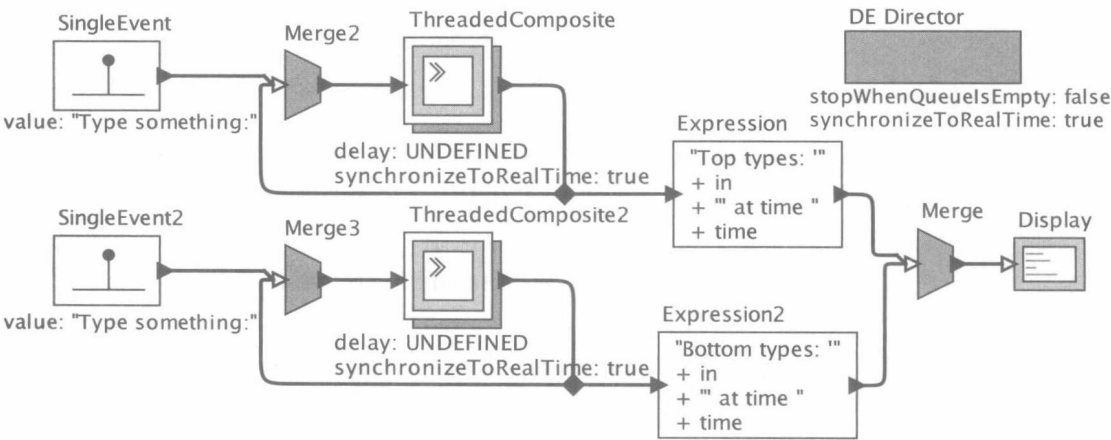


图 7-15 两个 InteractiveShell 实例在 ThreadedComposite 角色的不同线程中执行的模型

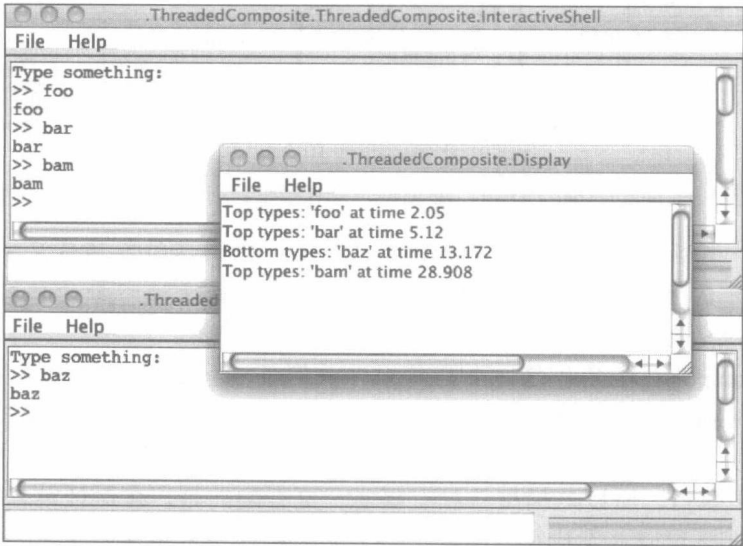


图 7-16 图 7-15 中一个模型的执行结果

通过拖拽 ThreadedComposite 的两个实例来创建该模型，然后把 InteractiveShell 的实例放在 ThreadedComposite 实例中，ThreadedComposite 角色的图标就变得与 InteractiveShell 一样，带有深灰色的阴影。

在这个模型中，InteractiveShell 的两个实例分别在 DE 指示器独立的两个线程中异步执行。当模型运行时，有 3 个线程在执行。在等待用户输入时，运行 InteractiveShell 角色的线程阻塞。当用户进行输入内容并按回车键时，InteractiveShell 角色产生一个输出，使其包含该角色的 ThreadedComposite 产生一个输出。

该模型还有许多其他细节。ThreadedComposite 角色的 delay 参数设置为 UNDEFINED，这样无论当前时间为多少，ThreadedComposite 角色都会将当前模型时间作为输出事件的时间戳。因此输出事件的时间戳是不确定的。

如果 delay 参数设置为 τ ，那么输出事件将模型时间为设置 $t + \tau$ ，其中 t 为点火输入事件的时间模型。这使得输出时间戳是确定性的。但是，这仍旧限制了并发性。当把 delay 设置为 τ 时，在当前模型时间为 $t + \tau$ 时该模型将阻塞。否则，ThreadedComposite 在阻塞前将尝试产生带有时间戳的输出事件。

另一个细微的影响是，指示器的 synchronizeToRealTime 参数设置为 true。这可确保“当前模型时间”不会比实际时间走得快。因此，在图 7-16 的输出跟踪中，报告时间可以看作从执行开始到用户输入所测得的时间。这就赋予了时间戳一个物理意义。模型中的不确定性也是合乎常理的，因为用户输入的时间相当不确定（这个不是由模型指定的）。

第三个细节是指示器的 stopWhenQueueIsEmpty 参数设置为 false。默认情况下，在没有事件需要执行时 DE 指示器将停止运行。但是在这个模型中，因为用户执行输入事件可以在稍后出现。因此，在队列为空时并不希望模型停止运行。

ThreadedComposite 角色为多线程中并发的执行模型提供了一种机制，并且该机制比直接使用线程更确定和更可控。直接使用线程非常困难（Lee, 2006）。ThreadedComposite 包含的角色不需要是原子角色，它可以是任意复杂的复合角色。事实是其包含的角色在独立的线程中执行，使得各个角色可以停止运行以便等待 I/O，为了提高性能它也能在多核机器上并行运行。该角色及其使用方面的更多内容由 Lee (2008b) 详述。

7.3.4 调度局限性

撰写本书时，Ptolemy II 中的 DE 指示器近似实现了 Lee and Zheng (2007) 所描述的语义。虽然，在理论上，所有模型都能够被准确的语义实现所执行，但当前的实现还不能执行所有的模型。

例 7.14 仔细观察图 7-17 中的模型。该模型在反馈回路中有一个不透明的复合角色。该复合角色中的 clock 输出不依赖于输入。因此，在任意给定的时间戳，在不知道输入端口是否有事件的情况下，该复合角色应该能够在 clock 输出端口产生一个事件。然而，试图执行这个模型会导致如下异常：

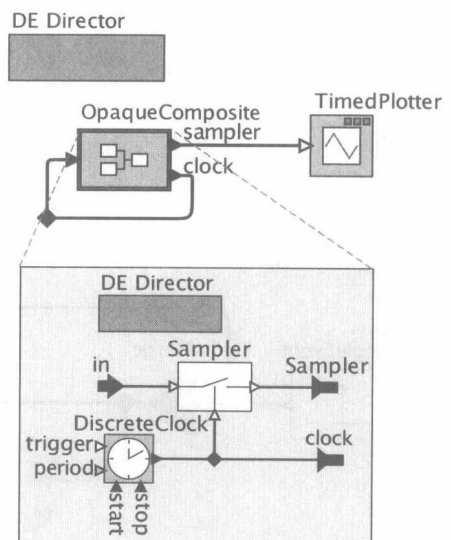


图 7-17 一个在理论上能够执行，但是 DE 目前不能够实现的模型

```

IllegalActionException: Found a zero delay loop containing
OpaqueComposite
in FixedPointLimitation

```

在当前的实现中，在每个时间戳（模型时间和微步）内，DE 指示器最多点火角色一次，并且当它点火那个角色时，需确保该时间戳的所有输入事件都是可用的。因此，图 7-17 中的复合角色永远不会被点火，因为指示器无法确定在当前时间戳有没有输入事件出现，直到它在那个时间戳可以点火复合角色。注意这个问题不能通过在反馈回路（见练习 1）中增加一个 0 延迟的 TimeDelay 角色来解决，但它可以通过使用具有固定点语义的指示器来解决，详见参考例子：FixedPointNoLimitation。

7.4 芝诺 (Zeno) 模型

当时间停止推进时可以创建一个如下所述的 DE 模型。

例 7.15 假设在图 7-11 中省略 BooleanSwitch，无条件反馈这些令牌。那么时间就不再增加，只有微步增加。

其中模型时间停止推进并只有微步增加的模型叫作抖动芝诺模型。DE 指示器将微步定义为 Java int 型，所以随着微步的增加最终将会导致溢出，从而导致指示器报告异常。

在时间推进但是不超过限定值的情况下，也可以创建芝诺模型。

例 7.16 图 7-18 是一个芝诺模型的例子。该模型使用 SingleEvent 角色（见第 7 章补充阅读：时钟角色）点火反馈回路。该反馈回路包括一个 TimeDelay 角色（详见第 7 章补充阅读：时间延迟），它的延迟值设置为 $1/n^2$ ， n 从 1 开始，令牌在回路中每循环一次 n 增加一。因此，延迟将接近 0，在时间到达 2.0 前这个模型可以产生无穷多的事件。

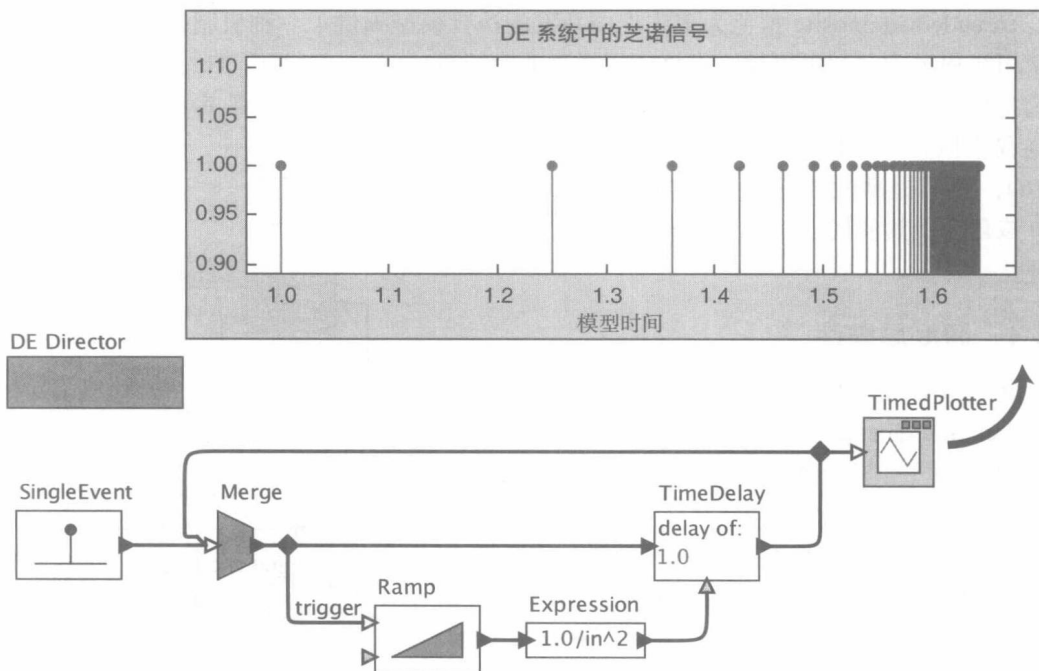


图 7-18 芝诺模型的例子，其中模型时间停止增加

在 DE 中，时间的精确度有限（见 1.7.3 节），所以芝诺模型最终变成抖动芝诺模型。当

时间增量降到时间分辨率以下的，时间就停止推进。然而，前面例子中的特殊模型在时间停止推进前就失效了，因为当 n 变得足够大时，计算 $1/n^2$ 时的浮点错误产生一个负值，使得 TimeDelay 角色报告一个异常。

7.5 其他计算模型与 DE 的组合

DE 模型可以有效地与其他计算模型组合。下面将给出一些有用的组合。

7.5.1 状态机和 DE

如图 7-1 所示，DE 模型中的角色可以由状态机来定义。这样的状态机可以在没有外部激发事件的情况下初始化反馈回路，如下例所示。

例 7.17 图 7-19 是一个包含单一 FSMActor 的简单 DE 模型。在该例中，初始状态是一个允许转移（条件的值为 true），它使得 FSMActor 在执行开始时就点火，同时产生一个输出。这个输出反过来又初始化反馈回路。这个角色不需要输入事件来触发第一次点火和初始化反馈。

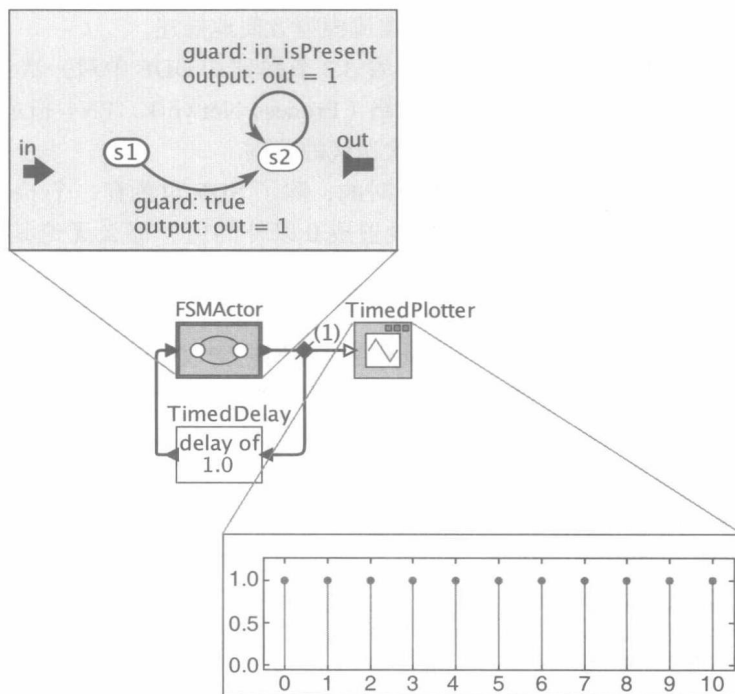


图 7-19 一个包含 FSM 的简单 DE 模型

在第一次点火后，这个角色处于状态 s_2 ，传出转移的条件是 $in_isPresent$ 。所以，接下来的点火需要输入事件。最终结果就是一个以一个时间单位分开的无界事件序列，如图 7-19 所示。

FSM 可以使用 `timeout` 函数（见表 6-2）直接控制时间来实现同样的效果，如图 7-20 所示。

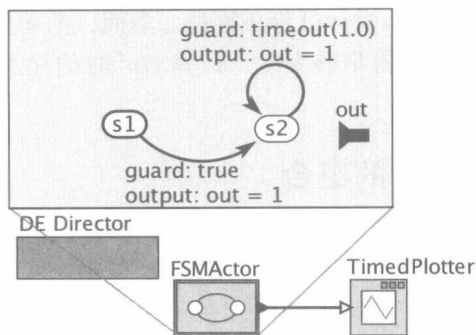


图 7-20 一个与图 7-19 具有同样行为的 DE 模型，但是它包含一个使用 `timeout` 函数来控制时间的 FSM

7.5.2 数据流和 DE 组合

在 DE 模型中可以使用数据流指示器。例如，SDF 指示器在 DE 模型中就很有用。如果 DE 模型中的复合角色里面包含 SDF 指示器，那么每当输入端口接收到一个事件时，内部数据流模型就会执行一次完整的迭代。当整个 DE 系统模型包含复杂的没有涉及计时事件的计算时，这个方法就很有效，并且也能够用数据流模型方便地描述。

DDF 指示器也可以用在 DE 模型内，尽管 3.2 节提到在 DDF 中对一次迭代进行控制比在 SDF 中困难。在 DE 内使用数据流进程网络（Process Network, PN）指示器没什么意义，就像在 4.1 节中讨论的那样，它很难确定一次迭代的范围。

第 3 章描述的数据流指示器一般是不计时的，除了 SDF 角色有一个 `period` 参数外。该参数在 DE 模型中很有用。如果这个参数设置成 0 以外的值，那么无论是否有事件输入，SDF 子模型都会在 DE 中周期性地执行。这个方法可以用来设计更复杂输出模式的时钟角色，而不是简单地使用指定的 `DiscreteClock`。然而，注意子模型只有在 `period` 的倍数时刻才运行，而不是有输入事件就马上运行。如果在某些周期倍数中没有足够的输入来运行一次完整的迭代，那么 SDF 模型就不在那时点火。而且，如果提供输入事件比 SDF 子模型处理它们的速度快，它们就在内存中排队等候，最后可能耗尽内存。练习 2 就是这样一个例子。

把 DE 模型置于 SDF 模型（或其他数据流模型）内没多大用处。DE 子模型希望在它的内部角色确定的模型时间点火，但是 SDF 模型只在 `period` 的倍数条件下点火。因此，这类复合模型通常会抛出一个如下异常：

```
IllegalActionException: SDF Director is unable to fire CompositeActor
at the requested time: ... . It responds it will fire it at: ...
in .DEwithinSDF.CompositeActor.DE Director
```

然而，如果 SDF 模型自己内嵌在一个 DE 模型中，那么可以把 DE 子模型放在 SDF 模型内，并把 SDF 模型的 `period` 参数设置为 0。这种情况下，SDF 指示器就会将点火请求授权给高阶的 DE 指示器，同时忽略自身的时间推进。

7.6 无线和传感器网络系统

在 DE 域中创建无线域（wireless domain）以便支持对无线网络进行建模。在无线域中，信道模型调节角色间的通信，并且视觉语法（即模型的图形化表示）不要求组件之间的有线连接。在无线域中模型的可视化表示比在其他 Ptolemy II 域中更重要，因为图标位置形成无

线系统的二维地图。在屏幕上图标的位置、它们之间的距离、它们之间的物体都会影响它们的通信。

例 7.18 图 7-21 中模型的顶层包含一个 WirelessDirector、两个 WirelessComposite 实例和一个 DelayChannel。WirelessComposite1 有一个输出端口，WirelessComposite2 有一个输入端口。尽管这些端口都不直接相连，但是它们可以相互通信。每一个端口有一个参数 outsideChannel，通过它通信的对象标记无线信道，在该例子中是 DelayChannel。

在这个简单的例子中，DelayChannel 组件对无线通信信道进行建模，发送消息的时延与模型中两个 wireless composite 图标之间的距离成正比。比例常数由 propagationSpeed 参数（距离 / 时间的任意单位）给出。在这个例子中，两个 Wireless Composite 图标大概相距 175 个单位，所以距离除以 100，传播速率大约为 1.75 个时间单位的时延。

在该模型中，WirelessComposite1 是事件的分散源（sporadic source）。分散源是随机源，其中事件与事件之间有一个固定时间下限。在该例中，下限由 Server 角色给出（详见第 7 章补充阅读：队列和服务器角色），随机性由 PoissonClock 角色给出。

模型中的 WirelessComposite2 作为时间函数简单描绘接收的事件。第一个事件由 WirelessComposite1 在时间 1.0 发出，WirelessComposite2 在大约 2.75 时接收。由此图可看出，两个事件之间的距离都不会小于一个时间单位。

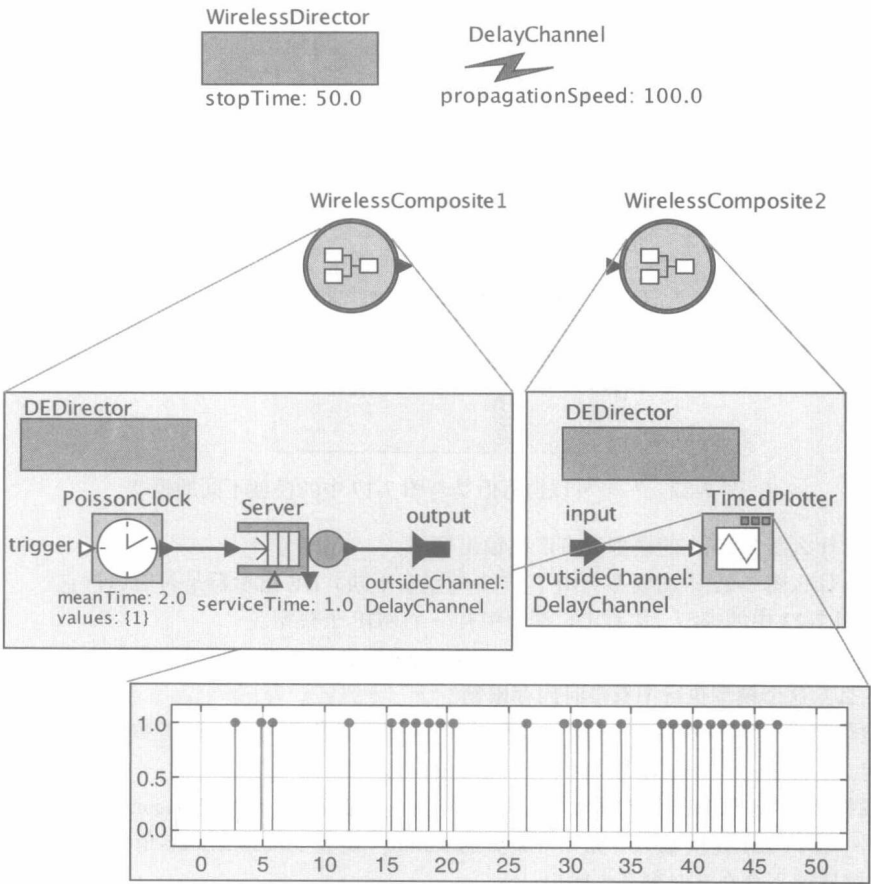


图 7-21 一个无线系统模型

除了对信道延迟建模外，无线域也可以对能量损耗、干扰、噪声以及阻塞建模。还可以对定向天线增益、移动发射机和接收器建模。详见 Baldwin et al. (2004) 和 Baldwin et al. (2005)，以及 Ptolemy II 软件包中的演示例子。

7.7 小结

DE 域为建模离散计时行为提供了坚实的基础。掌握它的使用需要对时间模型、并发和反馈有充分的理解，本章对此都有涉及。该域的关键特征在于：事件的执行顺序都是确定性的。在事件是并发情况下也不例外。

练习

1. 仔细观察图 7-22 中的模型。与图 7-17 中的模型不同，因为在反馈回路中有 TimeDelay，DE 指示器可以运行这个模型。

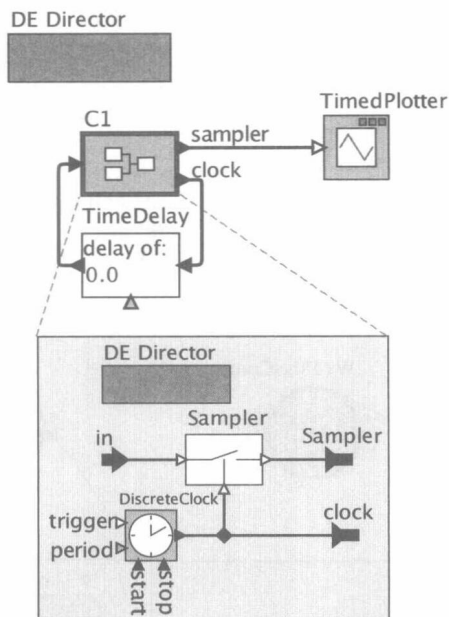


图 7-22 一个可以执行但是与图 7-17 中的模型不同的模型

- (a) 解释为什么图 7-22 中的模型没有产生输出事件。
 - (b) 解释为什么图 7-22 中的模型与图 7-17 中的模型不同，DE 指示器是否能运行它？
2. 仔细观察图 7-23 中的模型。它在 DE 模型中有一个 SDF 子模型。
 - (a) 假设 SDF 指示器的 period 参数是 1.5，DiscreteClock 的 period 是 1.0。该 CompositeActor 能输出什么？这个模型执行中有没有内存限制？
 - (b) 求能够产生图 7-24 中的 SDF 指示器和 DiscreteClock 角色的 period 参数值。解释为什么该输出确定。
 3. 这个问题探究 FSM 与 DE 模型组合的一些细节方面。构造由一个由 PoissonClock 组成的 DE 模型，它点火一个给 FSM 提供输入（见第 6 章）的 Ramp。设置 PoissonClock 角色的 fireAtStart 参数为 false，这样就不会在 0 时刻产生输出。
 - (a) 构建一个 FSM，即使在 0 时刻没有输入事件时，它也可以产生输出。
 - (b) 修改 FSM，当它接收到一个输入事件时，就在输入事件的模型时间产生两个输出。第一个输

出必须有与输入事件一样的值。第二个输出必须出现在一个微步后，且应该是输入事件值的两倍。

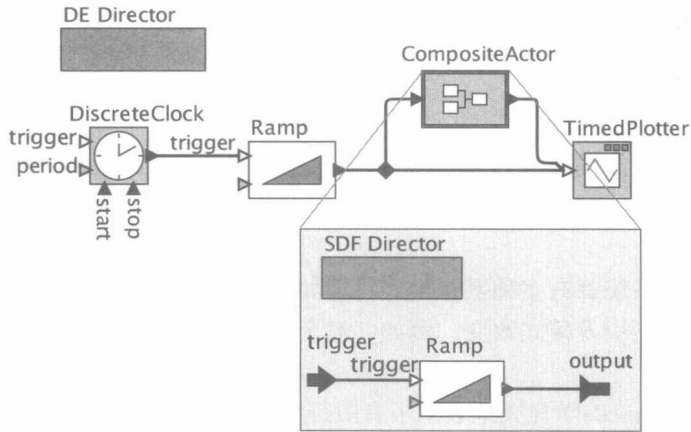


图 7-23 在 DE 模型内的 SDF 子模型的简单例子

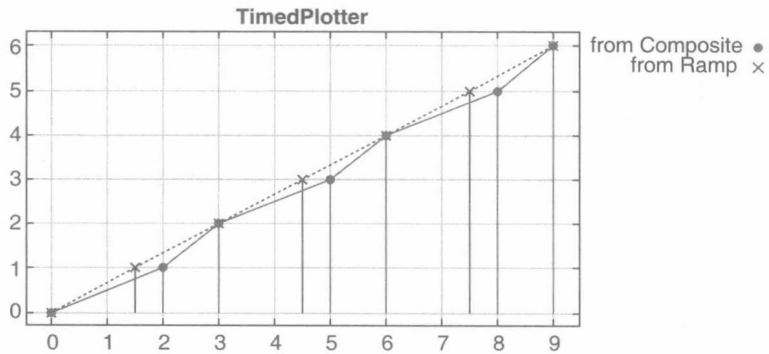


图 7-24 图 7-23 中模型产生的结果图

模态模型

Thomas Huining Feng、Edward A. Lee、Xiaojun Liu、Stavros Tripakis、
Haiyang Zheng 和 Ye Zhou

大部分有意思的系统都有多种操作模式。诸如用户输入、硬件错误、传感器数据等的内部或者外部事件都可能引发模式改变。例如，汽车中的发动机控制器在停止和行驶的时候就有不同的行为。

模态模型（modal model）显式地表示了有限行为（或者模式）的集合以及管理这些行为或模式之间转移的规则。这些转移规则通常通过有限状态机（Finite State Machine, FSM）来获取。

在 Ptolemy II 中，ModalModel（模态模型）角色用来构建模态模型。ModalModel 是一个分层角色，它同复合角色类似，但是它具有多个细化而复合角色只有一个细化。每个细化都是对行为的某个模式的具体化，而状态机决定哪个细化在给定时刻是有效的。ModalModel 角色比第 6 章提到的 FSMActor 更通用。FSMActor 不支持状态细化。模态模型使用与第 6 章描述一样的转移和条件（guard），并在此基础上进行了一些扩展。

例 8.1 图 8-1 中的模型说明一个通信通道有两种操作模式：干净的（clean）和有噪声的（noisy）。模型中有一个具有两种状态（clean、noisy）的 ModalModel 角色（标记为“Modal Model”）。在 clean 模式中，模型不做改变地将输入传送给输出。在 noisy 模式下，在每一个输入令牌中增加一个高斯随机数。顶层模型提供了一个由 PoissonClock 角色产生的事件信号，该角色根据泊松过程随机产生事件。（在泊松过程中，事件之间的时间独立同分布这些呈指数分布的随机变量。）该模型执行的样本，是 Signal Source 角色提供的一个输入正弦波，结果如图 8-2 所示。

该例有 3 个确定的计算模型（MoC）。在顶层，随机事件的计时行为通过 DE 域捕获。下一层使用 FSM 捕获模式的变化。第三层使用 SDF 捕获样本数据的不计时处理。

创建模态模型的过程如图 8-3 所示。为了在 Vergil 中创建模态模型，从 Utilities 库中拖拽一个 ModalModel 角色，并通过端口来进行配置。打开模态模型角色，增加一个或多个状态和转移。为了创建转移，按住 Control 键（Mac 中为 Command 键）、单击、从一种状态拖拽到另一种状态。为了增加细化，右击一个状态，选择 Add Refinement。你可以选择 Default Refinement 或者 State Machine Refinement。例 8.1 使用了前者，它在每一个细化过程中需要一个指示器和处理输入到输出数据的角色。后者产生一个分层 FSM，如第 6 章所述。

8.1 模态模型的结构

模态模型的一般结构如图 8-4 所示。模态模型的所有行为都由状态机管理，每一个状态就是一个模式（mode）。在图 8-4 中，每一个模式由一个椭圆符号（与状态机中的状态相似）表示。但是必须指出：模式是一种特殊的状态。与普通的状态不同，模式有模式细化（mode refinement），它是不透明的复合角色，用来定义模式的行为。图 8-1 中的例子展示了两个细

化，每个都是处理令牌输入产生输出令牌的 SDF 模型。

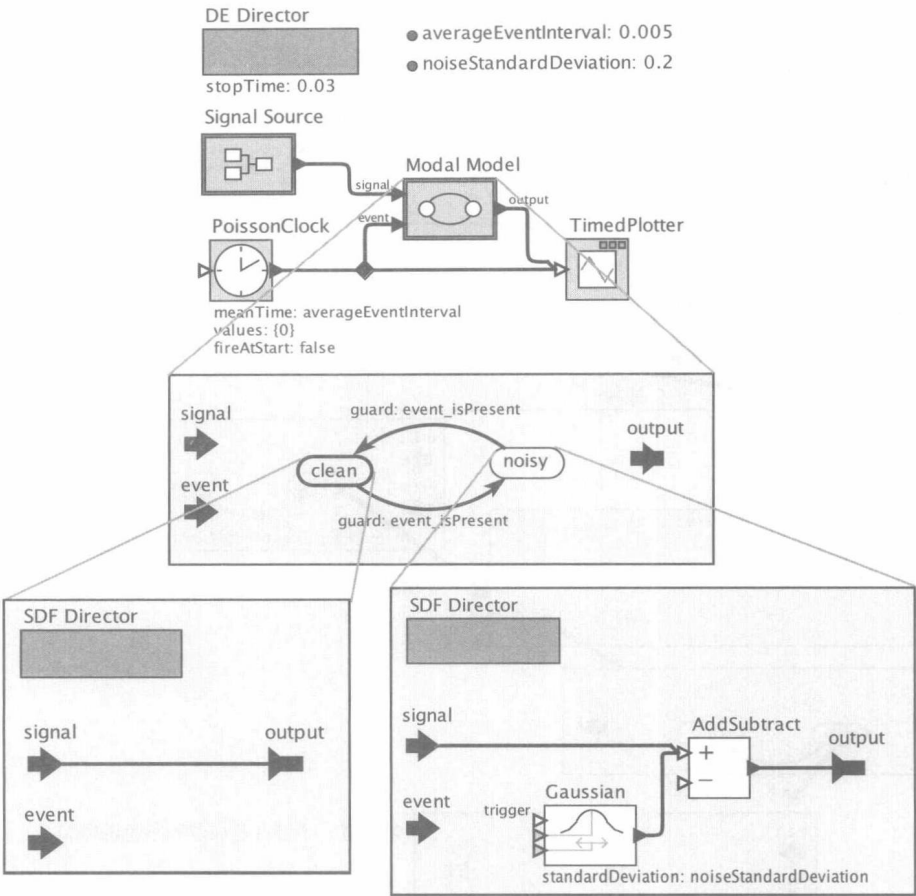


图 8-1 一个简单的模态模型，它有一个正常的（清晰的）操作模式（它不做任何改变地将输入传送到输出），它有一个错误模式（它会增加高斯噪声（Gaussian noise））。它根据 PoissonClock 角色决定的随机时间在模式之间进行转换

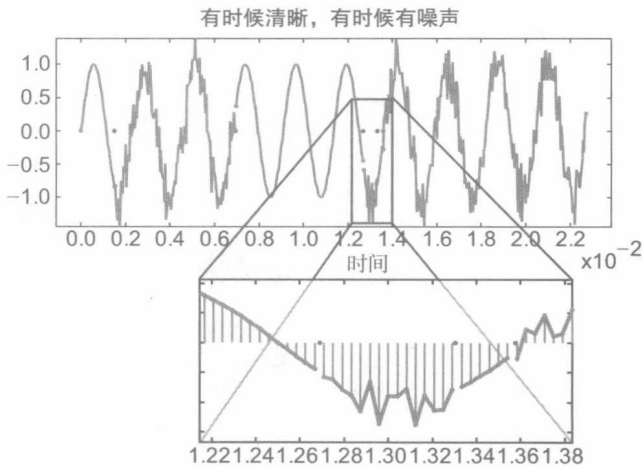


图 8-2 图 8-1 中模型产生的结果图

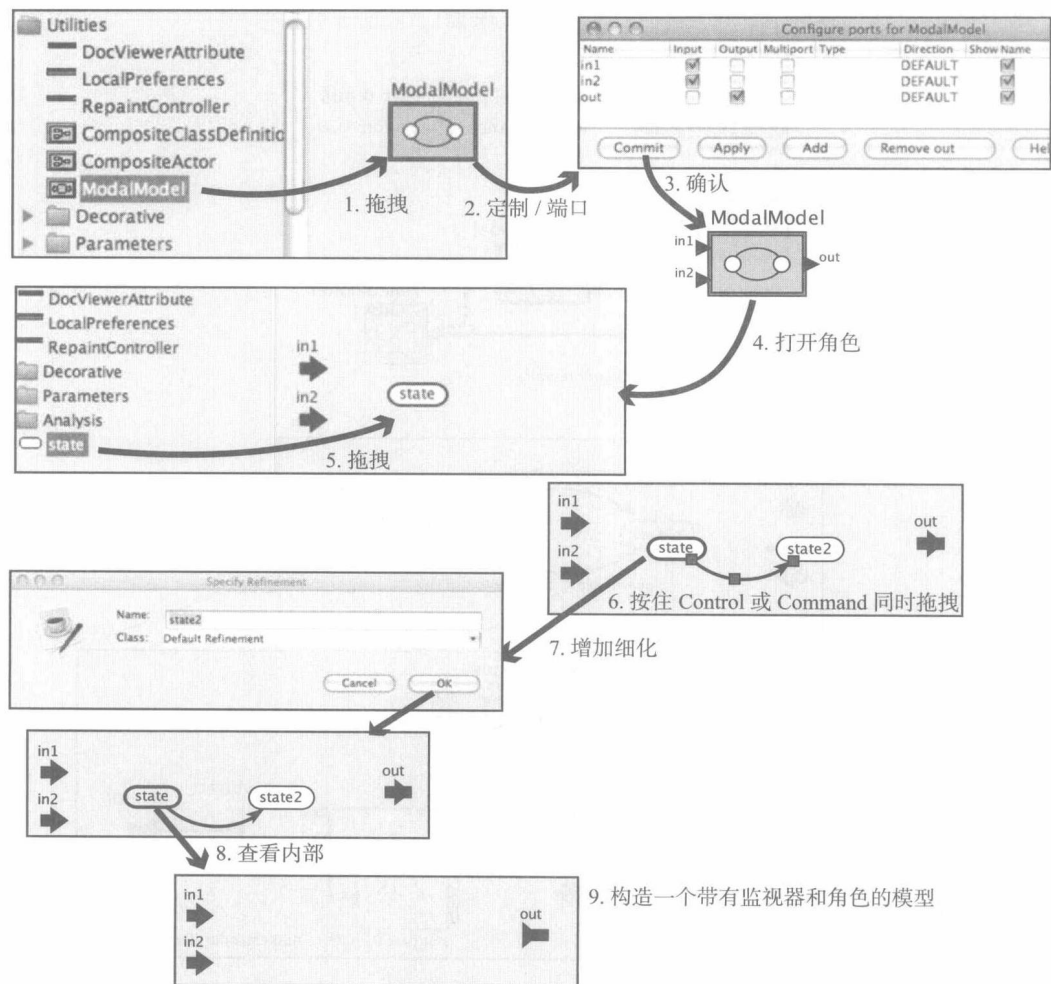


图 8-3 如何创建一个模态模型

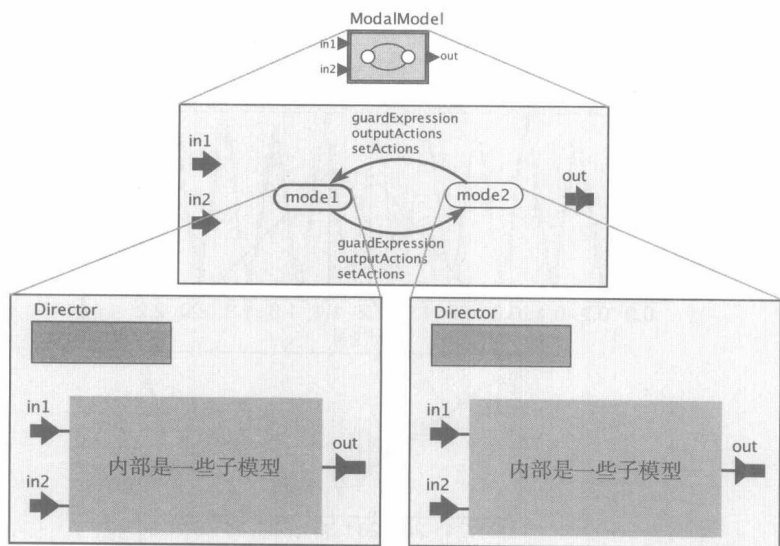


图 8-4 有两个模式的模态模型的通用模式，每个模式都有自己的细化

模式细化必须包含一个指示器，这个指示器必须与管理模态模型角色执行的指示器相匹配。图 8-1 中的例子在每一个模式中都有一个 SDF 指示器，每一个模态模型外都有一个 DE 指示器。SDF 通常用在 DE 内，所以这个组合是有效的。

与有限状态机中的状态类似，模式通过代表转移的弧线相连接，每个转移都带有条件，这些条件用来指定转移发生的时间。

例 8.2 图 8-1 中，转移的条件为 `event_isPresent`，当 `event` 输入端口有事件时其值为 `true`。由于这个输入端口与 `PoissonClock` 角色相连，所以这个转移将在一个随机时间发生，其用一个指数随机变量来控制转移之间的时间。

图 8-5 是图 8-4 结构的变体，其中两个模式共享同一个细化。当不同模式的行为只是参数值不同时，这个就非常有用。例如，习题 2 创建了图 8-1 的另一种变体，这里的 `clean` 细化与 `noisy` 细化只有 `Gaussian` 角色的参数值不同。为了创建模型（多个模式有相同的细化），给其中一个状态增加一个细化，然后给该细化命名（默认情况下，建议的细化名与状态名相同，但是用户可以自己定义细化名）。然后，对另一个状态，选择 `Configure`（或者简单地双击该状态）而不选择 `Add Refinement`，指定 `refinementName` 参数值为细化名。这样，两个模式就会有相同的细化。

另一种变体就是一个模式拥有多种细化。这个效果可以通过多次执行 `Add Refinement` 或者为 `refinementName` 参数指定一个用逗号分隔的细化名列表来实现。这些细化将按它们加入的顺序执行。顺序可以通过调用（或者双击）状态的 `Configure` 并编辑用逗号分隔的细化名列表来改变。

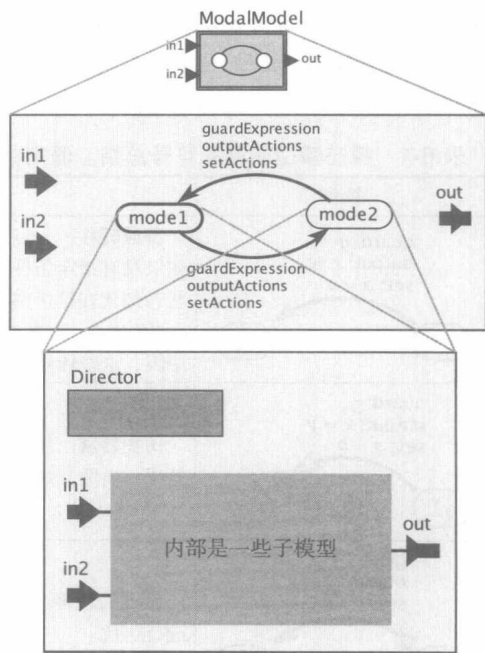


图 8-5 图 8-4 模式的变体，其中两个模型共享同一个精化

进一步探索：模态模型的内部结构

在 Ptolemy II 中，每个对象（角色、状态、转移、端口、参数等）最多有一个容器。但是在模态模型中，两个状态可以共享同一个细化，这似乎违反了通用规则。

最主要的不同是，`ModalModel` 角色实际上是一个专用的复合角色，它包含一个 `FSMDirector` 实例，一个 `FSMACTOR` 和一些复合角色。每个复合角色都可以是 `FSMACTOR` 角色任何状态的细化。`FSMACTOR` 是控制器，其意义在于决定哪个时间内哪个角色是活动的。`FSMDirector` 保证输入的数据传递给了 `FSMACTOR` 和所有活动的模式。在 6.3 节介绍的分层 FSMs 中使用相同的结构。

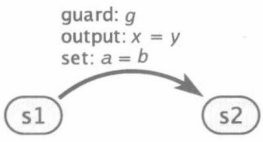
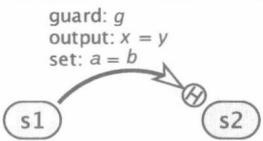
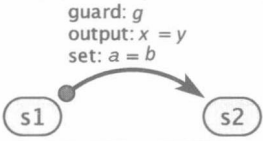
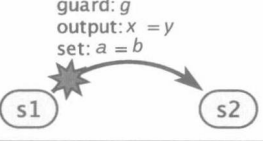
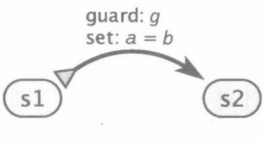
在 Vergil 用户界面中不显示这种结构，当用户在 `ModelaModel` 中执行一次 `Open Acotor` 命令，界面将直接转入 `FSMACTOR` 控制器，而不会显现包含 `FSMACTOR`、`FSMDirector` 及其细化的分层内容。然而当具体查看某一状态时，用户界面将会进入到

该结构的上一级，并打开所选状态的全部细化。这种构造方式主要是希望在系统展示与用户便利间取得平衡。

8.2 转移

表 6-1 中所有的转移类型在模态模型中都可以使用，它们在模态模型中表达的意义与原 FSM 一致。表 6-3 所示用来解释分层 FSM 的转移类型，但是在不是 FSM 的细化中意义稍微有点不同。一个 FSM 状态的细化可以是任意的不透明复合角色（包含一个指示器）。当某些细化是 FSM，某些细化是其他模型时，它们也可以混合使用。这些转移更通用的含意在本节中详述，在表 8-1 对此进行了总结。

表 8-1 模态模型转移和符号总结。假定状态细化是任意 Ptolemy II 模型，每一个都有一个指示器

符号	描述
	<p>普通转移。一旦点火，首先点火源状态的细化，然后，如果条件 g 为 <code>true</code>（或者如果没有指定条件），那么 FSM 将选择转移。它在输出端口 x 产生一个值 y，重写源状态细化在这个端口上产生的值。一旦发生转移（在后点火阶段），角色就设置变量 a 的值为 b，再次重写细化已经赋给 a 的值。最终，初始化状态 $s2$ 的细化。出于这个原因，这些转移有时也叫作复位转移</p>
	<p>历史转移。历史转移和普通转移相似，除了当进入状态 $s2$ 时，状态的细化没有初始化。当然，第一次进入 $s2$ 时，细化已经被初始化了</p>
	<p>抢占式转移。如果当前状态是 $s1$，且条件是 <code>true</code>，那么 $s1$ 的状态细化在转移前不进行迭代</p>
	<p>差错转移。如果状态 $s1$ 的任意细化都抛出一个异常或模型错误，且条件是 <code>true</code>，那么执行这个转移。这个转移的输出和赋值动作会涉及几个特别的变量：<code>errorMessage</code>、<code>errorClass</code> 和 <code>errorCause</code>，这些变量将在 8.2.3 节中解释</p>
	<p>终止转移。如果状态 $s1$ 的所有细化在后点火阶段返回 <code>false</code>，且条件是 <code>true</code>，那么执行这个转移。注意，由于只有到了后点火阶段才知道该转移是可执行的，所以这个转移不能产生输出。对于大多数域来说，后点火阶段太迟而不能产生输出。而且，因为该转移只有在后点火阶段才有可能发生，所以它的优先级比所有的转移都低，包括默认转移</p>

8.2.1 复位转移

默认情况下，转移就是复位转移，也就是说当转移发生时，目标状态的细化已经被初始化了。如果该细化是 FSM，如 6.3 节所描述，就意味着 FSM 的状态将被设置为初始状态。如果初始状态有细化状态机，那么这些状态机也被设置为它们的初始状态。实际上，复位转移的机制很简单：调用细化的初始化方法。这使得细化内的所有部件都被初始化。

例 8.3 如图 8-1 所示，转移是否是历史转移不重要，因为这两个状态的细化中都没有

状态。这个模型（Gaussian 和 AddSubtract）中的角色没有状态，所以将它们初始化不会改变它们的行为。

然而，在图 8-6 的例子中，Ramp 角色有状态。该例子展示了一个历史转移，它产生了图 8-7a 所示的结果。在这种情况下，当重新进入某个状态时，Ramp 角色就从它们上次离开的地方继续计数（而不是重新开始）。

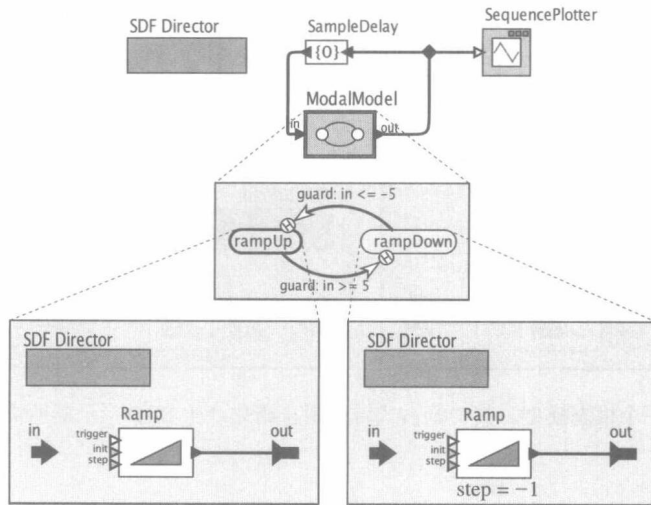


图 8-6 一个模态模型，根据转移是复位转移还是历史转移来决定它的行为

另一方面，如果我们将转移改为复位转移，结果就会如图 8-7b 所示。每次使用一个转移，将 Ramp 角色初始化（随着细化的重置），同时它们再次开始从 0 开始计数。

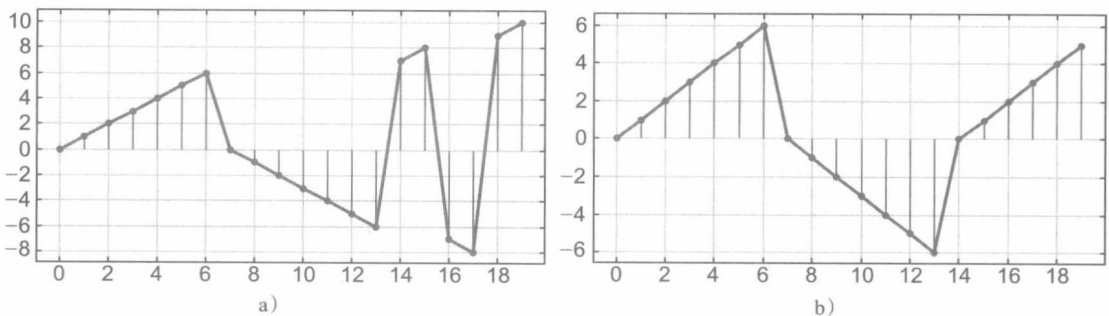


图 8-7 a) 使用历史转移执行图 8-6 中的模型所产生的结果图，b) 改为复位转移产生的结果图

8.2.2 抢占式转移

对于一般的模态模型，抢占式转移的工作模式与分层 FSM 中的一样。如果条件可用，细化就不执行，其结果就是细化不产生输出。

例 8.4 对图 8-6 模型进行修改使得转移都是抢占式的，得到图 8-8 所示的模型。这意味着，当条件的值为 true 时，当前状态的细化不会产生输出。在这个特别模型中，迭代根本没有产生输出，这违背了 SDF 原则，该原则希望所有的点火都产生一个固定的、预定数量的令牌。因此，就产生了一个图中所示的错误。这个错误可以通过在转移中产生一个输出或

使用不同的指示器来修正。

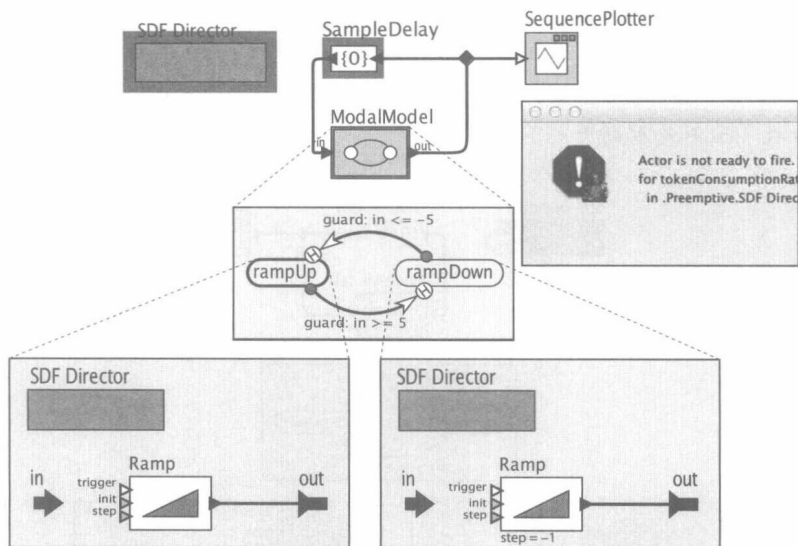


图 8-8 一个模态模型，其中抢占式转移阻止精化产生 SDF 指示器所需要的输出

8.2.3 差错转移

在执行细化时，可能出现抛出异常的错误。默认情况下，异常将导致整个模型终止执行，这并不总是大家所希望的。也许模型能够从错误中完美地恢复。为了实现这一目标，Ptolemy II 状态机包括了**差错转移**（error transition），在执行当前状态细化产生一个错误时，点火该转移。差错转移也可能有条件、输出动作、赋值动作。当使用输出动作时，需要一些警告，然而，因为如果错误出现在细化执行的后点火阶段，那么就会因为太迟而不能产生输出。但是，大多数错误将在点火阶段出现，因此大部分情况下这并不会成为一个问题。

例 8.5 如图 8-9 所示，这是一个包含差错转移的模型。与例 8.6 一样，该模型包含一个允许用户输入任意文本信息的 InteractiveShell 角色。在这个模型中，将用户输入的内容传送到 ExpressionToToken 角色，该角色解析用户输入，并用 Ptolemy II 表达式语言（详见第 13 章）解释文本。当然，用户可能输入无效的表达式，这将使 ExpressionToToken 抛出一个异常。

在 FSM 中，listening 状态有一个差错转移自循环。差错转移的起始端标记有深灰色星。当 listening 状态的细化抛出一个异常且它的条件（如果有的话）为 true 时，就点火差错转移。在这种情况下，条件确保该转移不超过 3 次。3 次之后，转移就再也不会被点火了。

在图的下方展示了这个模型执行的例子。这里，用户首先输入一个有效表达式“2*3”，它产生结果 6。接着，用户输入一个无效表达式“2*foo”。无效是因为模型中没有以“foo”命名的变量。这就点火了一个异常，这个异常将被差错转移捕获。

在这个简单的例子中，差错转移只是简单地返回到相同的状态。事实上，该转移也是一个历史转移，所以细化不能再被初始化。这对差错转移很危险，因为异常可能会使得细化进入不一致的状态。但是，这个例子是没有此问题的。如果这是一个复位转移，那么在捕获

异常后 InteractiveShell 将会被初始化。这将导致 shell 窗口被清空，与用户互动的历史记录被删除。

在执行第 4 个无效表达式 “3*baz” 时，差错转移条件的值不再为 true，所以没有捕获到异常。这将使得模型停止运行，出现一个异常提示窗口，如下图中下方位置所示。

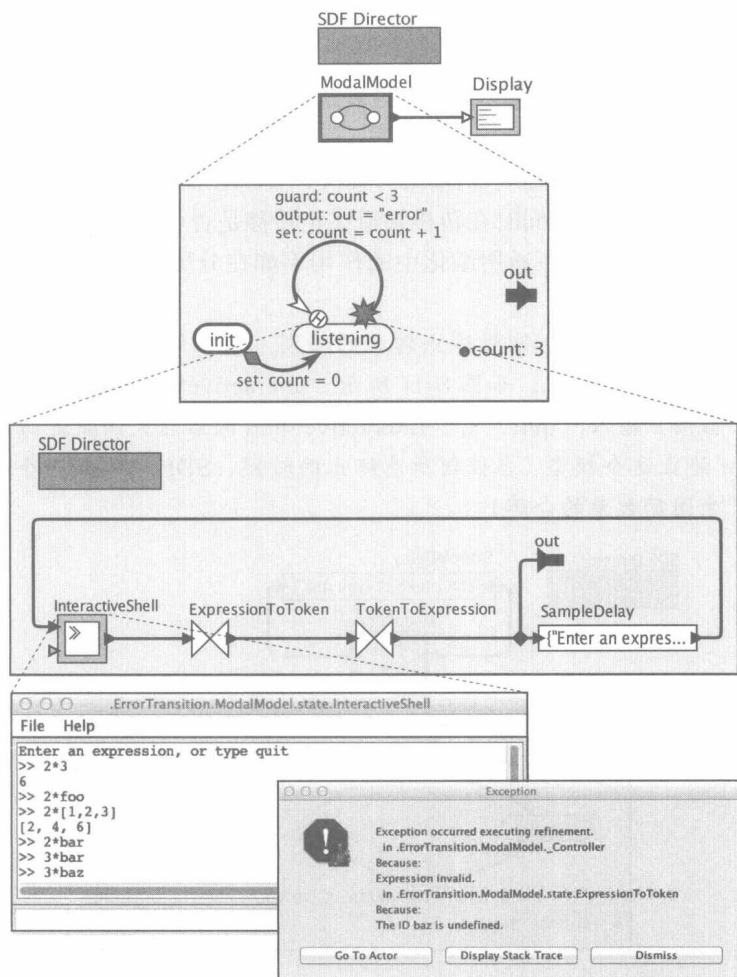


图 8-9 包含差错转移的模态模型

在模型中，差错转移提供一个强大的机制以便从错误中恢复。使用差错转移时，有两个变量需要设置，它们可能在转移的条件、输出动作或赋值动作中使用：

- **errorMessage**：错误信息（字符串）。
- **errorClass**：抛出异常的类型名（字符串）。

另外，有些异常还需要设置第 3 个变量：

- **errorCause**：引发异常的 Ptolemy 对象。

对于上面的例子，**errorCause** 变量将会涉及 **ExpressionToToken** 角色。这是一个 **ObjectToken**，通过它可以调用一些方法，如条件中的 **getName**，或者转移的输出或赋值动作（详见第 14 章）。

8.2.4 终止转移^①

当状态细化是通用 Ptolemy 模型而不是分层 FSM 时，终止转移（termination transition）的动作将十分不同。这样的终止转移会在当前状态的所有细化终止时发生，但是对于通用 Ptolemy 模型，我们不可能知道模型是否在执行的后点火阶段前结束。因此，如果当前状态中的细化至少有一个是默认细化（与状态机细化相比），那么：

- 不允许终止转移产生输出。且
- 终止转移的优先级低于其他转移，包括默认转移。

这个约束的原因很微妙。首先，在许多域（例如，SR、Continuous）中，后点火阶段太迟而不能产生输出。下游角色将看不到输出。其次，要求在执行点火阶段中计算所有其他转移（非终止转移）的条件的值，同时在转移知道终止转移是否可用前选择转移。

由于这些约束，终止转移在通用细化中的作用不如在分层 FSM 中。然而有时候它们的确有用。

例 8.6 图 8-10 展示了一个使用终止转移的模型。这里的主要角色是 InteractiveShell，它打开一个供用户输入的对话框，如图 8-11 所示。InteractiveShell 要求用户输入，或者输入“quit”来终止。当用户输入“quit”时，InteractiveShell 的后点火函数返回 false，它使得下方的 SDFDirector 终止这个模型（当任何角色终止的时候，SDF 就终止一个模型，因为违背了 SDF 规则：产生固定数量的令牌）。

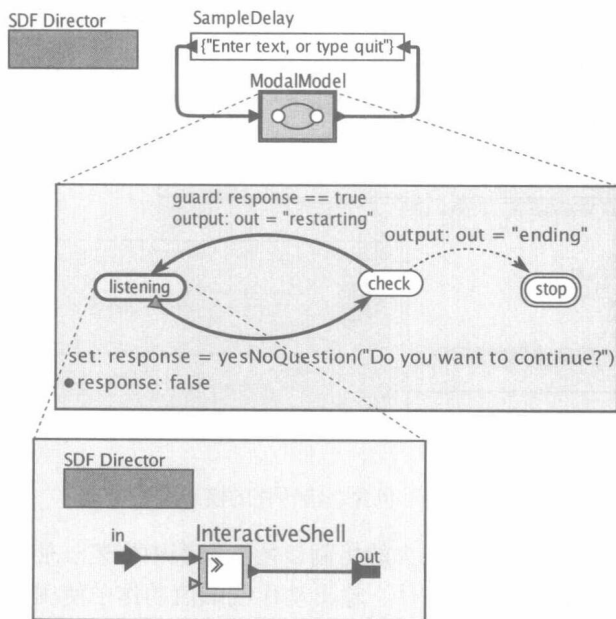


图 8-10 具有终止转移的模态模型

在 FSM 中，从 listening 到 check 的转移是一个终止转移，所以当用户输入“quit”时它就点火。这个转移有如下形式的动作：

```
response = yesNoQuestion("Do you want to continue?")
```

它调用 yesNoQuestion 函数从而弹出一个对话框来获得用户的选择，如图 8-11 所示（详见

① 终止转移非常专业化。第一次阅读本书的读者可以选择跳过这节。

表 13-16 了解 yesNoQuestion 函数)。如果用户选择“yes”，那么转移将 response 参数设置为 true，否则设置为 false。因此，在下次迭代中，FSM 要么将使复位转移回到初始化 listening（监听）状态，打开另一个对话框，要么终止转移，返回最终状态。转移到最终状态将导致顶层 SDF 指示器终止。

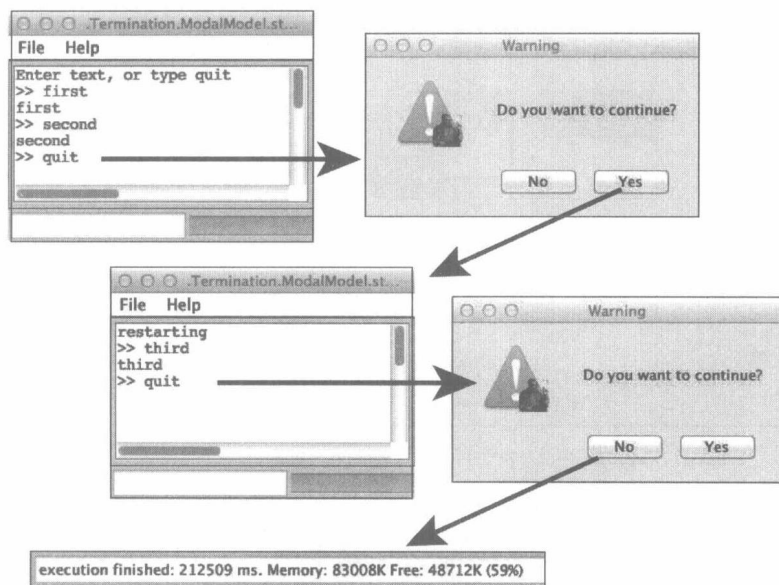


图 8-11 图 8-10 中模型的执行过程

8.3 模态模型的执行

ModalModel 的执行与 FSMActor 的执行类似。在 fire 函数中，ModalModel 角色：

- 1) 读取输入。
- 2) 计算出当前状态传出抢占式转移的条件值。
- 3) 如果没有抢占式转移，那么角色
 - (a) 点火当前状态的细化（如果有）；
 - (b) 计算出当前状态向外的非抢占式转移的条件值；
 - (c) 选择一个条件的值为 true 的转移，假设优先选择抢占式转移；
- 4) 执行选定转移的输出动作。

在 postfire 函数中，ModalModel 角色

- 1) 如果它们被点火了，则后点火当前状态的细化。
- 2) 执行选定转移的赋值动作。
- 3) 当前状态修改为选定转移的目标状态；且
- 4) 如果转移是复位转移，初始化目标状态的细化。

ModalModel 角色在它的 fire 函数中没有进行持续的状态改变，所以只要细化指示器与角色相同，模态模型就可以用于任何域中。然而，在每一个域中它的行为可能有差异，特别是在使用定点迭代或者使用非确定性转移的域中。在 8.4 节中，将讨论在各种域中模态模型的使用。

注意，在计算非抢占式转移的条件值前，点火状态细化。这个顺序的结果是，条件可能

涉及细化的输出。因此，当前细化对输入的响应决定了是否进行转移。聪明的读者有可能已经注意到本章的图，FSM 的输出端口与正常的输出端口（注意图 8-1 和图 8-4 中的 output 端口）不同。在 FSM 中，输出端口实际上既是输出也是输入。它为 FSM 中的所有角色提供这两种功能。当前状态细化的输出也是 FSM 的输入，检测器将影响这个输入。

例 8.7 图 8-12 展示了图 8-10 模型的一个变体，其中包括引用了一个细化输出的条件。这个条件定义对用户输入“hello”的响应，如下图所示。

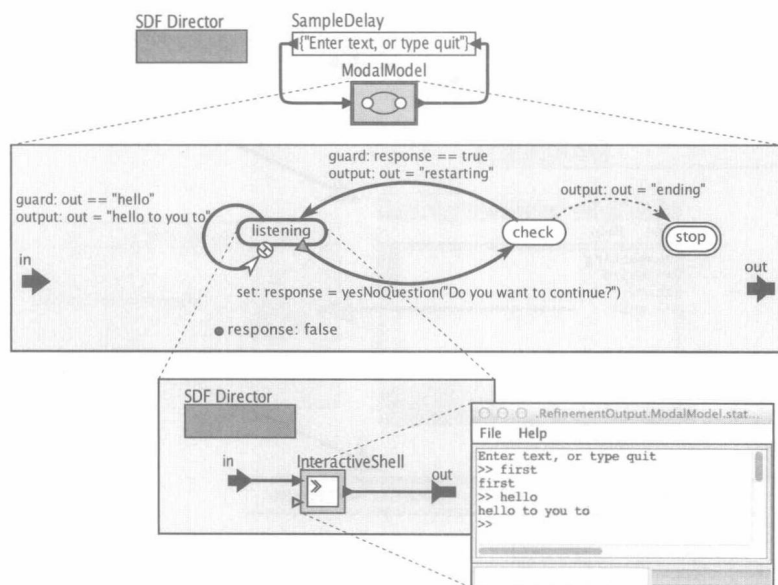


图 8-12 图 8-10 模型的一个变体，它包括一个引用细化输出的条件

上面的例子说明，当前状态细化和转移的输出动作都可以在同一个输出端口产生输出。由于 FSM 的执行是严格按顺序的，所以 ModalModel 的输出端口产生的结果也是确定的。在点火阶段总是将最后得到的值写入输出。甚至有一连串的立即转移，每个转移都经过具有写入相同输出端口的细化的状态，每个转移也都写入相同输出端口。这些写通常以已经定义好的顺序进行，只有最后一个写是模态模型对外可见的。

8.4 模态模型和域

目前，模态模型例子主要以简单的方式来使用 SDF 和 DE 域。对于 DE 的例子，如图 8-1 所示，当至少一个输入端口有事件时，模态模型就将点火。有些输入是 `absent`，一个输入事件的存在（或缺失）可能会引发转移。模态模型可能或不可能在每一个输出端口产生输出；如果没有产生输出，那么输出就为 `absent`。在 DE 中使用模态模型的精妙之处在于它涉及时间的流逝，这将在 8.5 节详叙。

然而，在其他域中模态模型的作用没有那么简单。我们将在本节针对一些细节进行讨论。

8.4.1 数据流和模态模型

目前的 SDF 例子都是同构 SDF，其中每个角色消耗并产生一个令牌。当这些例子中的模态模型点火时，模型的所有输入都是 `present`，都恰好包含一个令牌。模态模型的点火导致每一个输出端口产生一个令牌，除了在图 8-9 中的错误阻止了输出令牌的产生外。

模态模型可以谨慎地与多速率 SDF 模型一起使用，如下例所述。

例 8.8 在图 8-13 所示的例子中，由于有 SequenceToArray 这一角色，每一个状态的细化都需要 10 个样本才能点火。模型的状态在求 10 个输入样本的平均值和计算其中最大值这两个状态间转变。ModalModel 每次点火时都执行当前细化的一次迭代，这里处理 10 个样本。正如在结果图中看到的，当输入是正弦波时，10 个样本的平均序列产生另一个正弦波，当输入是最大值时将发生变化。

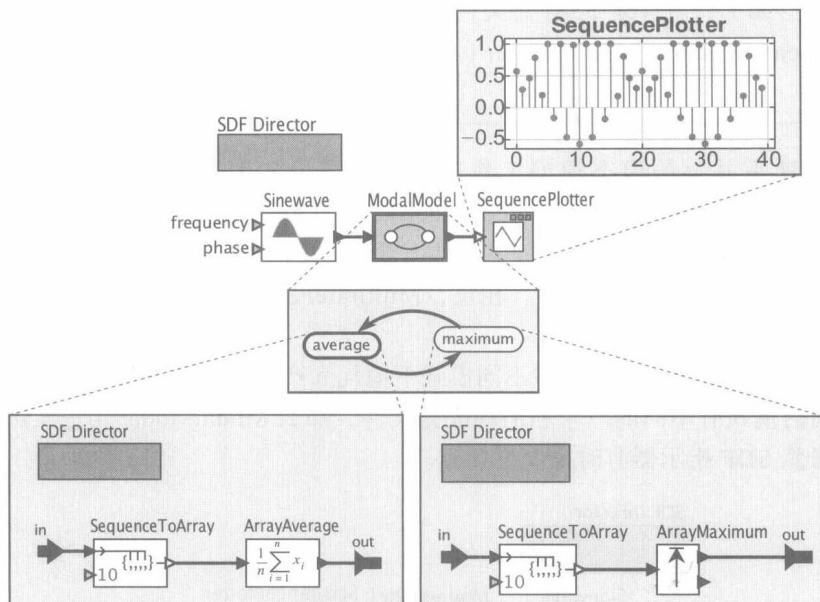


图 8-13 一个 SDF 模型，为了点火，ModalModel 需要其输入有多个令牌

进一步探索：并行和分层状态机

并行和分层 FSM 的一个早期模型是状态机 (Statecharts)，它由 Harel (1987) 提出。Harel 提出了状态与 (and state) 的概念，这里一个状态机可以同时处在状态 *A* 和 *B*。仔细检查，状态机模型是 SR 计算模型下的分层 FSM 的一个并行部分。因此状态机 (大致) 类似于结合了分层 FSM 和 Ptolemy II 中 SR 指示器的模态模型。另外，在模型细化中使用 SR 指示器来管理并发角色，每一个都是状态机，提供了 Statecharts 的一个变体。用软件工具实现的 Statecharts 叫作 Statemate (Harel et al., 1990)。

Harel 的工作给大家带来了许多灵感，产生了许多模型的变体 (von der Beeck, 1994)。一种变体被 UML (Booch et al., 1998) 采纳。一个特别优雅的版本是 SyncCharts (André, 1996)，它为 Esterel 同步语言提供了一个可视化语法 (Berry and Gonthier, 1992)。

状态机的同步复合的一个重要特性是，它可以像状态机一样对部件的复合进行建模。实现它的简单机制将导致状态机的状态空间是单个状态空间的叉积。后来又提出了更复杂的机制，例如接口自动机 (de Alfaro and Henzinger, 2001)。

混合系统 (见第 9 章) 也可以视为模态模型，其并发模型是一个连续的时间模型 (Maler et al., 1992; Henzinger, 2000; Lynch et al., 1996)。在通常的设想中，混合

系统将 FSM 和常微分方程 (ODE) 组合在一起, 每个 FSM 的状态都与 ODE 的一个特别配置联系起来。各种用于描述仿真和分析混合动力系统的软件工具已经开发出来了 (Carloni et al., 2006)。

Girault et al. (1999) 首先说明 FSMs 可以与许多并发计算模型分层地组合起来。这种组合称之为 **charts* 或 *starCharts*, 这里 *star* 表示通配符。许多活跃的研究项目继续探索并发状态机的表示变体。例如, BIP (Basu et al., 2006) 使用会话交互来组成状态机。Alur et al. (1990) 针对并发 FSM 语义问题以及其中的许多复杂问题。给出了一个完美的研究, Prochnow and von Hanxleden (2007) 描述了并发状态机的可视化编辑的复杂技术。

为了让多速率并存的模态模型工作, 需要将产生和消耗的信息从细化传播给顶层 SDF 指示器。这样, 就必须改变 ModalModel 角色的 *directorClass* 参数, 如图 8-14 所示。ModalModel 的默认指示器不关心令牌的产生与消耗, 因为它与任意 Ptolemy II 指示器一起工作, 而不是仅仅与 SDF 一起工作。相反, MultirateFSMDirector 就是与 SDF 合作, 在层次之间传递产生和消耗的信息。

在特定情况下, 甚至允许细化在不同模式下消耗和产生的配置文件不同。但是要格外注意, 如果不同的模式有不同的产生和消耗配置文件, 那么 ModalModel 角色实际上就不再是 SDF 角色。虽然 SDF 指示器有时也允许这点。

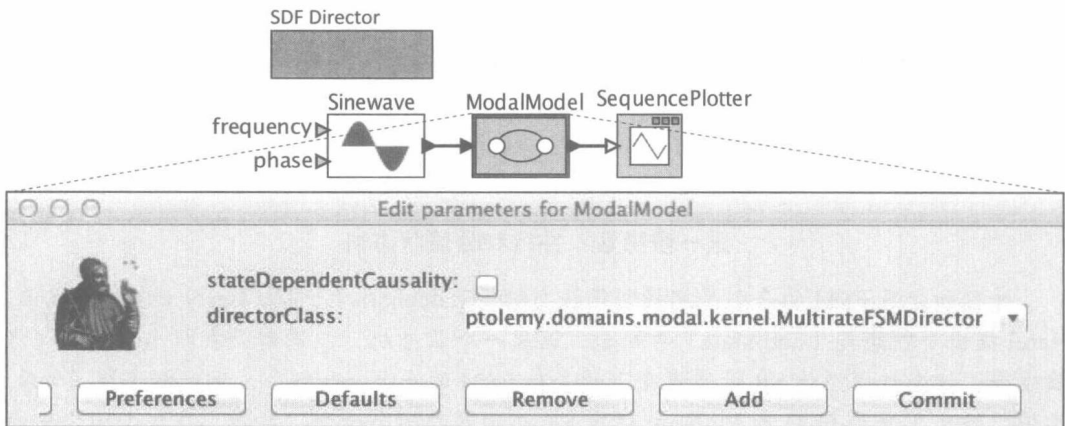


图 8-14 为了使得 ModalModel 成为可以在输入输出端产生非均匀消耗的 SDF 角色, 因此, 使用了特定的 MultirateFSMDirector

例 8.9 在图 8-13 中, 可以改变 SequenceToArray 角色的参数, 这样它们在两个细化中就不一样。在幕后, 每次发生转移时, 顶层的 SDF 都会查看产生和消耗配置文件的变化, 并计算一个新的调度表。

当角色的产生和消耗配置文件发生改变时, 需要 SDF 指示器来重新进行调度计算, 这是一个主要的差别。另外, SDF 指示器只在执行完一个完整的迭代后才重新计算 (见 3.1.1 节)。如果产生和消耗配置文件在一个完整迭代的中间发生改变了, 那么 SDF 就不能完成该完整的迭代。

由于输入令牌不足或者内存容量不够有可能导致角色不能被点火的问题。

因子改变时，HDF 指示器计算一个新的调度，新的调度输出下一个斐波那契数字。

例 8.11 图 8-16 展示了另一个有趣的例子。在该例子中，两个递增的数字序列合并为一个递增序列。在初始化状态，ModalModel 从每个输入各消耗一个令牌，并在它右上方的输出端口输出较小的序列，在它右下方的输出端口输出较大的序列。当然，较小的序列是合并序列中的第一个令牌。将较大的序列反馈到 ModalModel 中一个名为 previous 的端口。

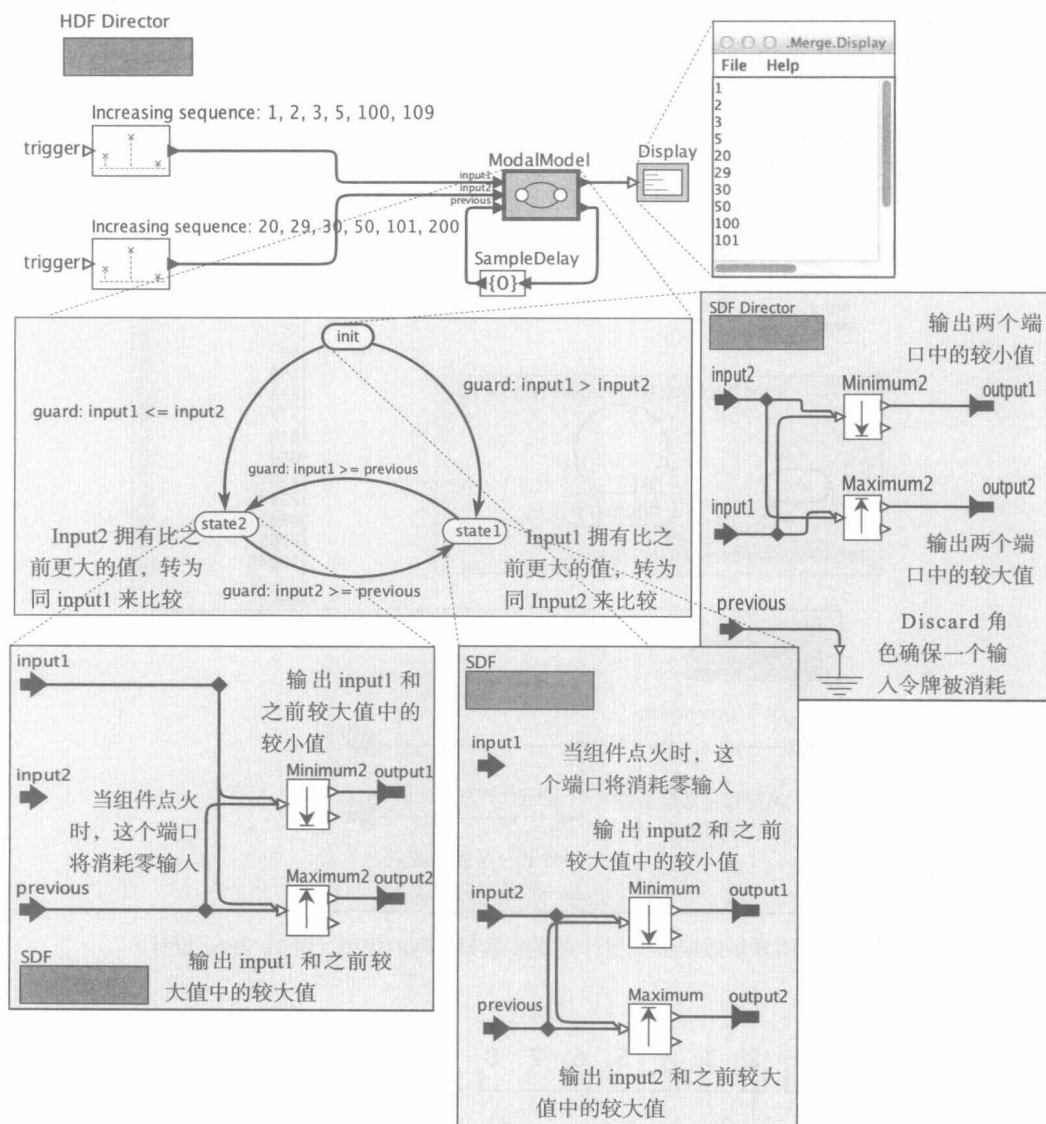


图 8-16 一个异步数据流模型，它合并两个数字递增序列为一个数字递增序列（由 Ye Zhou 和 Brian Vogel 设计）

如果较大的输入来自 input1，那么 FSM 转换到 state1。这个状态的细化不是从 input1 读取一个令牌（它的消耗参数是 0）。相反，它从 input2 读取一个令牌并将它与反馈回来的值进行比较。

如果最初较大的输入来自 input2，那么 FSM 就转移到 state2，它从 input1 读，并将它

与反馈回来的值进行比较。

这些例子证明：HDF 允许动态的改变消耗和产生的速率。在每类情况下，模态模型关于产生和消耗配置文件由当前状态细化内的模型决定。

HDF 计算模型由 Girault et al. (1999) 提出，HDF 指示器的原作者是 Ye Zhou。它有许多有趣的特性。与 SDF 一样，HDF 模型可以对死锁和有界缓冲区进行静态分析。但其 MoC 比 SDF 更灵活，因为 Moc 允许依赖数据的产生和消耗的速率。然而，为了使用这个模型，模型构建者需要对完整迭代 (complete iteration) 有着深刻的理解，因为当 FSM 中发生转移时这个概念是受限的。

8.4.2 同步响应和模态模型

第 5 章所描述的 SR 域可以通过很多有趣的方式来使用模态模型。与 DE 或者数据流比较，主要的差别在于 SR 模型有反馈回路，它需要通过迭代来收敛于一个定点。6.4 节列举了一个使用 FSM 反馈回路的例子。

需要注意的问题是，当 ModalModel 角色在 SR 域中点火时，它的有些输入为未知 (unknown)。这和输入为缺失 (absent) 不一样。当输入为未知时，那么无法确定该输入是存在 (present) 还是缺失 (absent)。

为了使得模态模型能够用于反馈回路，即使当某些输入为未知时模态模型也能确定输出就显得十分重要。确定输出意味着能确定输出是存在还是缺失。如果输出为存在 (present)，就随意给它赋个值。但是输入变为已知 (known) 时，模态模型必须十分谨慎地确定它此前对输出的判断没有错误。这个约束可确保角色是单调的。

如果 ModalModel 角色被一些未知的输入点火，那么它必须区分“确定一个转移不可用”和“不确定一个转移不可用”。如果条件中有未知的输入，那么它就不确定一个转移是否可用。实际上，这使得确定“输出为缺失”很有挑战性。例如，仅确定在当前状态下没有转移可用是不够的。相反，模态模型必须确定每一个可能使输出存在的转移是已知不可用的。

该约束与立即转移链有着微妙的关系，因为从当前状态转出的所有转移链都需要考虑该约束。如果在这个转移链中的任一转移中有一个条件不确定其值为真还是假，那么随后转移的可能输出都要被考虑。如果立即转移链中有状态细化，那么当所有输入都为已知时，很难断定是一个输出为缺失 (absent)。

出于这些细节考虑，不建议在反馈回路中使用那些没有确定性能输入，却要求能够确定输出的模态模型。因为这将导致模型很难理解，甚至识别正确的行为都成问题。

8.4.3 进程网络和会话

模态模型可以与进程网络 (PN) 和会话 (rendezvous) 结合使用，但只是一种简单的方式。当一个 ModalModel 角色点火时，它将读取每一个输入端口 (自顶向下顺序)，在每一个域中将引起所有角色阻塞直到输入可用。因此，模态模型总是从每个输入消耗一个令牌。而是否在输出上产生一个令牌，将由 FSM 决定。

8.5 模态模型中的时间

许多 Ptolemy II 指示器执行计时的计算模型。ModalModel 角色和 FSMActor 自己是不计时的，但是有许多支持它们在计时域中使用的特性。

目前讨论的 FSM 是反应式的 (reactive)，意味着只产生响应输入的输出。在计时域中，输入有时间戳。对于一个响应 FSM，输出的时间戳和输入的时间戳相等。这使得 FSM 对输入的响应看起来是即时的。

在计时域中，也可以定义自发的 FSM 和模态模型。一个自发的 FSM (spontaneous FSM) 或自发的模态模型 (spontaneous modal model) 可以在输入缺失时产生输出。

例 8.12 图 8-17 中的模型使用 6.2.1 节介绍的 timeout 函数，在条件表达式中每 1.0 个时间单位点火一个转移。这个自发的 FSM 根本没有输入端口。

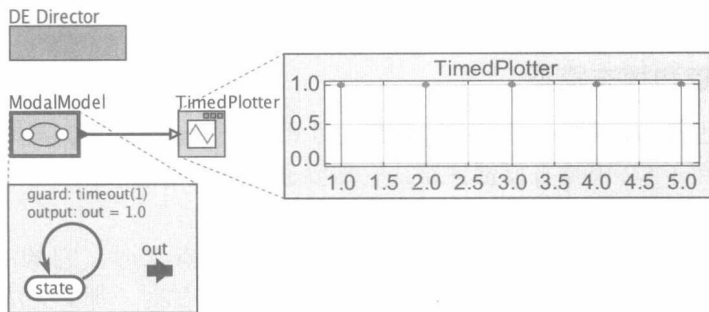


图 8-17 一个自发的 FSM，它在没有输入事件点火的情况下产生输出

例 8.13 图 8-18 中的模型每 2.5 个时间单位在两个模型中切换一次。在 regular 模式中，它每 1.0 个周期 (DiscreteClock 的默认输出值为 1) 产生一个均匀间隔的时钟信号。在 irregular 模式中，它使用 PoissonClock 产生间隔随机的事件，事件的平均时间设置为 1.0，值设置为 2。典型的运行结果如图 8-19 所示，有阴影背景的部分展示了两个模型使用的时间。ModalModel 的输出事件是自发的；响应输入事件对于它们来说不是必需的。

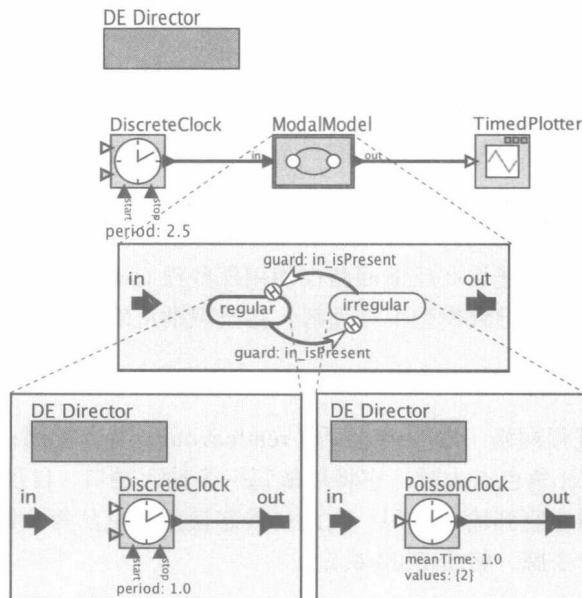


图 8-18 另一个自发的模态模型，可在没有输入事件触发的情况下产生输出事件

该例说明模态模型中时间使用的许多微妙性。在图 8-19 中，在时间 0 产生了 2 个事件：一个值为 1，另一个值为 2。为什么呢？初始状态是 regular，图 8-3 描述的执行策略说明了

在评估条件前点火初始状态的细化。点火产生 DiscreteClock 的第一个输出，其值为 1。如果改用抢占式转移，如图 8-20 所示，那么第一个输出事件将不会出现。

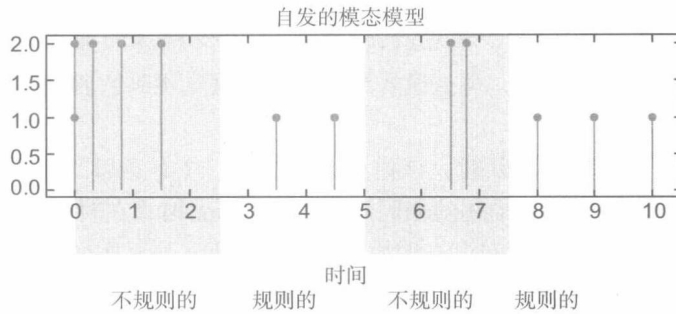


图 8-19 图 8-18 中模型的一次运行的输出

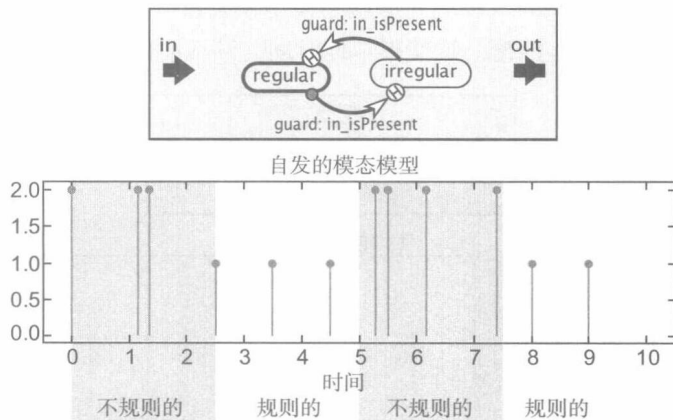


图 8-20 图 8-18 的一个变体，其中抢占式转移阻止 DiscreteClock 的初始点火

图 8-19 中的第二个事件（值为 2）在时间 0 产生，因为 PoissonClock 在执行开始时默认产生一个事件。这个事件在 ModalModel 的第二次迭代中产生，在进入 irregular 状态后。尽管该事件与第一个事件有相同的时间戳（都在时间 0 出现），但它们有定义良好的次序。值为 1 的事件先于值为 2 的事件出现。

如前描述，在 Ptolemy II 中，时间的值由一对数字 $(t, n) \in \mathbb{R} \times \mathbb{N}$ 表示而不是一个数字（详见 1.7 节）。第一个数字 t 叫作时间戳，这是一个实数（这是一个精度明确的量化实数）。时间戳 t 的单位为秒（或者其他的单位），是模型从执行开始时消耗的时间。第二个数字 n 叫作微步，它代表一系列出现在同一个时间戳的事件。在本书的例子中，第一个事件（值为 1）有一个标记 $(0, 0)$ ，第二个事件（值为 2）有一个标记 $(0, 1)$ 。如果我们设置 PoissonClock 角色的 fireAtStart 参数为 false，那么第二个事件就不会发生。

注意，在 regular 模式细化中的 DiscreteClock 角色的周期为 1.0，但是在时间 0.0、3.5、4.5、8.0 和 9.0 等产生事件。不是执行开始时间 1.0 的整数倍，为什么？

模型从 regular 模式开始，但是这里消耗 0 时间。它立刻转移到 irregular 模式。因此，在时间 0.0，regular 模式变为不活跃。当它不活跃时，它的本地时间不再向前推进。在全局时间 2.5 它变为活跃，但本地时间仍然是 0.0。因此，它还得再等一个时间单位，直到 3.5，

才产生下一个输出。

本地时间 (local time) 的概念对于理解计时模态模型很重要。很简单, 当模式不活跃时, 本地时间将停止。涉及时间的角色, 如 `TimedPlotter` 和 `CurrentTime`, 可以根据本地时间或者全局时间来确定响应时间, 如 `useLocalTime` 参数名描述的那样 (默认为 `false`)。然而, 如果没有角色使用全局时间, 那么模式细化就完全察觉不到它曾经挂起过。它好像感受不到时间已经流逝。

当 `irregular` 模式再次变为活跃时, 该模型输出的另一个有意思的性质是在时间 5.0 没有事件产生。这个行为来自于上面描述的相同规则。`irregular` 模式在时间 2.5 变为不活跃状态, 因此, 从 2.5 ~ 5.0, 它的本地时间没有增加。当它在时间 5.0 再次变为活跃时, 它继续等待适当的时间来从 `PoissonClock` 角色中产生下一个输出[⊖]。

如果要求在时间 5.0 产生一个事件 (当 `irregular` 模式变为活跃时), 那么可以使用一个复位转移, 如图 8-21 所示。在初始化时, `PoissonClock` 的 `initialize` 方法产生一个输出事件。复位转移使本地时间与环境时间 (环境时间就是模态模型所处的模型时间, 这些不同的时间值将在接下来的章节中讨论) 匹配。这使得本地时间与环境时间的间隔为 0。

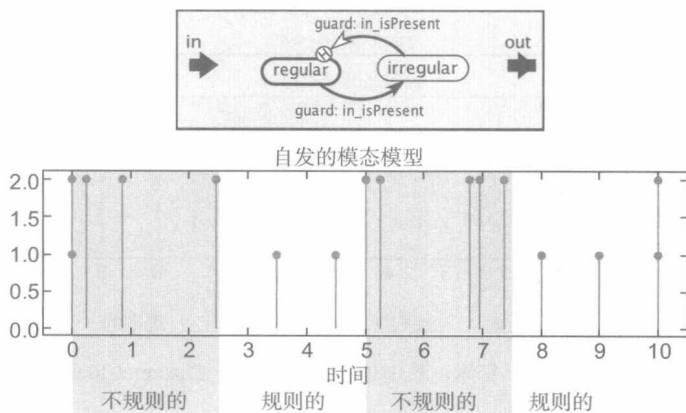


图 8-21 图 8-18 的另一个形式。当 `irregular` 模式重新活跃时, 一个复位转移引起 `PoissonClock` 产生事件

8.5.1 模态模型中的时间延迟

在模态模型中使用时间延迟可以产生许多有意思的效果, 如下例所示。

例 8.14 图 8-22 展示了一个模型, 该模型每隔一个时间单位产生一个事件计数序列。这个模型使用两种模式, 延迟与不延迟, 每隔一个事件延迟一个时间单位。在延迟模式中, `TimeDelay` 角色将事件延迟一个时间单位。在不延迟模式中, 直接将输入传送给输出而不延迟。这个模型的执行结果如图 8-23 所示。注意时间 2 产生值 0。为什么?

这个模型从延迟模式开始, 它接收第一个输入。这个输入的值是 0。然而, 在时间 0 后模态模式转移立刻从延迟模式变为不延迟模式。在时间 1 延迟模式再次变为活跃的, 但是它的本地时间仍然是 0。因此, 必须将值为 0 的输入延迟一个时间单位, 直到是时间 2。它的输出在时间 2 产生, 正好是在再次转移到不延迟模式之前。

⊖ 有趣的是, 由于泊松过程的无记忆性, 所以统计上变为活跃的下一个事件的时间与泊松过程的事件之间的时间相等。但这个事实与模态模型的语义没有什么关系。

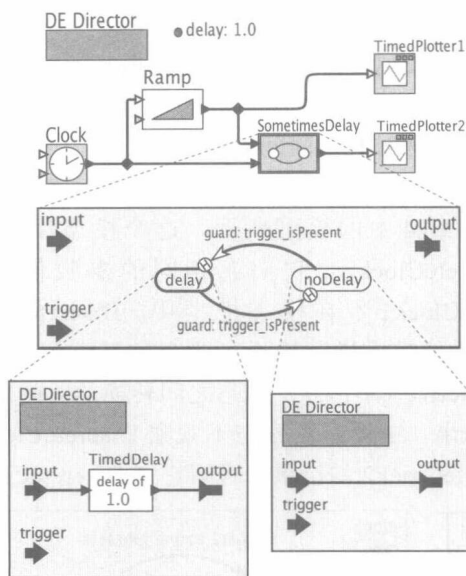


图 8-22 在输入延迟一个时间单位与不延时之间切换的模态模型

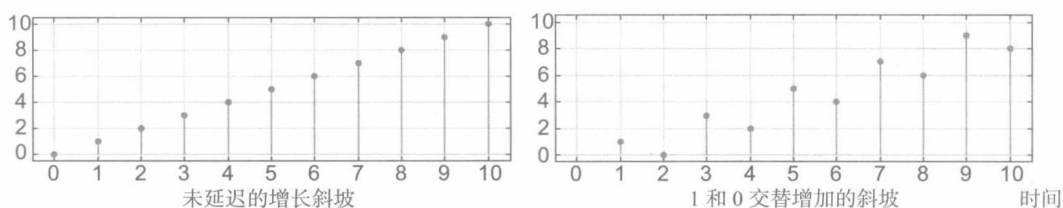


图 8-23 图 8-22 中模型的执行结果

8.5.2 本地时间和环境时间

如前面的例子所述，模态模型有很复杂的行为，尤其是在计时域中使用。深入思考 Ptolemy II 在实现模态模型所选用的设计原则是非常有用的。模型的主要思想是，在一部分时间内指定一部分系统有效。当它不活跃时，它是不是终止运行？时间继续吗？状态变化吗？这些问题都很难回答，因为行为的设计依赖于应用。

在模态模型中，有 4 个潜在的影响模型行为的不同时间：本地时间、环境时间、全局时间和实际时间。**本地时间**（local time）是模式（或其他本地角色）内的时间。**环境时间**（environment time）是包含模态模型的模式内的时间。**全局时间**（global time）是分层模型中最高层的模型时间。**实际时间**（real time）是计算机执行模型之外的挂钟时间（也就是物理世界时刻）。

在 Ptolemy II 中，指导原则是：当模式不活跃时，本地时间保持静止，环境时间（和全局时间）继续。所以不活跃模式是指处于假死状态。模式内的本地时间将通过**累计挂起时间**来延迟其环境时间，或是**延迟**那些没有减少的时间。

模式细化中的时间间隔最初是模态模型的开始环境时间和模式细化的开始时间之间的差值（一般这个差值为 0，但是它也可以是非 0 的，详见 8.5.3 节）。每次模式变为不活跃，时

间延迟就增加，但是在模式内，时间看起来就是连续的。

当事件通过分层边界进入或离开模式时，它就根据时间间隔来调节时间戳。也就是说，当模式细化产生一个输出事件时，如果那个事件的本地时间是 t ，那么在模态模型输出的事件的时间就为 $t+\tau$ ，这里的 τ 是累计挂起时间。

一个重要的发现就是，当子模型不活跃时，它与子模型接收输入而忽略它们的行为方式是不同的。这点如图 8-24 中的模型所示。这个模型说明了 DiscreteClock 的两个实例，DiscreteClock1 和 DiscreteClock2，它们有相同的参数值。DiscreteClock2 在模态模型 ModalClock 中，DiscreteClock1 不在模态模型中。DiscreteClock1 的输出由模态模型 ModalFilter 过滤，选择合适的输入进行输出。两个模态模型由相同的 ControlClock 控制，它决定 active（活跃）与 inactive（不活跃）状态之间转换的时间。图 8-24 显示了 3 种情况。上图是 DiscreteClock1 的输出。中图是观察与不观察 DiscreteClock1 输出之间转换的结果。下图是活跃与不活跃 DiscreteClock2 的结果，否则它与 DiscreteClock1 相同。

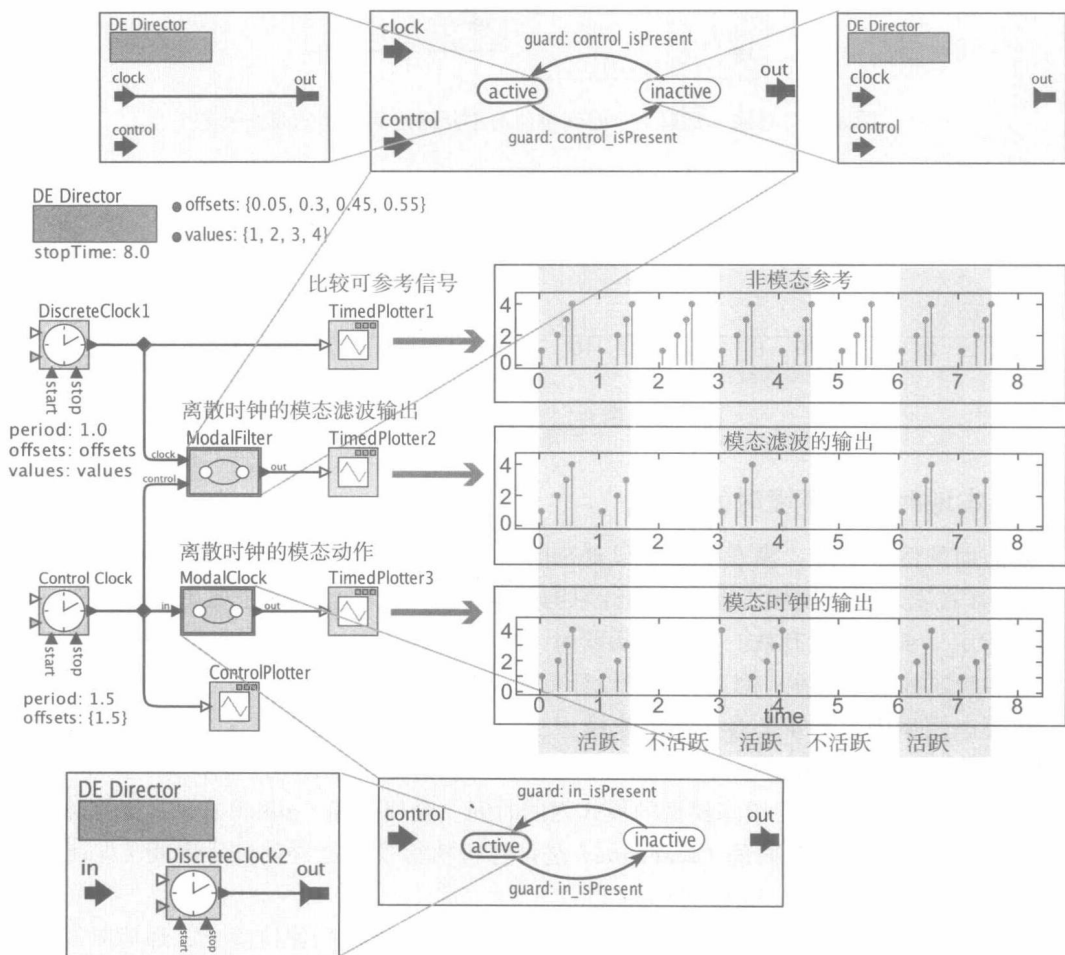


图 8-24 在模态模型中植入一个定时角色 DiscreteClock，它与对输出进行观察与不观察之间切换的模型是不同的

这个例子中的 DiscreteClock 角色用来循环产生序列值 1, 2, 3, 4。因此，除了能够计时外，这些角色都有状态，因为为了产生下一个输出值它们需要记住上一个输出值。当

DiscreteClock2 不活跃时，它的状态不会改变，时间不推进。因此，当它再次变为活跃时，它会从中断位置。

8.5.3 模式细化中的开始时间

通常，当执行一个计时模型时，我们希望它从开始时间到停止时间都按照指定的时间间隔执行。默认情况下，执行从时间 0 开始，但是指示器的 `startTime` 参数可以指定不同的开始时间。

然而，当 DE 模型在模式细化内时，默认情况下，子模型的开始时间就是初始化的时间。通常，它与上层模型（enclosing model）的开始时间一样，但是当使用复位转移时，那么子模型可以在任何时间被重新初始化。

当子模型被复位转移重新初始化时，在一个特定时间重新执行偶尔也是很有用的。这个功能可以通过改变 DE 指示器内的 `startTime` 参数的默认值（默认值为空，即初始化的时间）来实现。

例 8.15 图 8-25 说明了 `startTime` 参数的使用，它实现了一个可重置计时器（resettable timer）。这个例子是一个带有单一模式和单一复位转移的模态模型。DE 指示器内的 `startTime` 设置为 0.0，这样当每次发生复位转移时，子模型的执行会再次从时间 0.0 开始。

在这个例子中，PoissonClock 产生使复位转移发生的随机重置事件。这个模式细化有一个 SingleEvent 角色，它用来在时间 0.5 产生一个值为 2.0 的事件。如图 8-25 所示，在收到一个输入事件后，模态模型在 0.5 时间单元产生一个输出，除非它在这个 0.5 时间单位内接收到第二个输入事件。第二个事件重置定时器然后重新启动。因此，在时间 1.1 事件没有产生任何输出，因为在时间 1.4 事件重启了计时器。

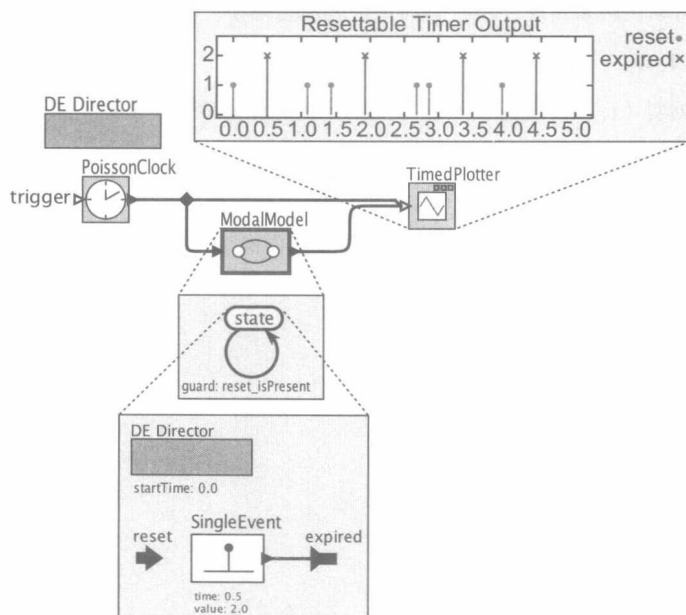


图 8-25 通过使用复位转移在时间 0 重新启动子模型来实现可重置定时器

当发生复位转移且目标模式细化有一个特定的 `startTime` 时，累计挂起时间就增加 t ， t 是上层模型的当前时间。复位转移发生后，本地时间与全局时间的时间间隔比之前发生的转

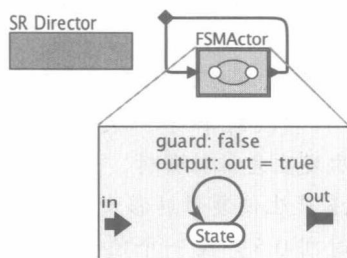
移的时间差大 t 。

8.6 小结

Ptolemy II 中的 FSM 和模态模型提供了一个建立复杂模态行为的有效方法。本章使用了一些实例来解释如何使用它们。本章提供了一个开头。希望深度研究的读者可以通过运行 Java 类来检查这些文档以寻找实现这些机制的 Java 类。在运行 Vergil 时通过右击并选择 Documentation 可以找到更多有用文档。

练习

1. 在下面这个模型中，唯一的信号（从 FSMActor 的输出返回到它的输入）在所有的节拍都有值 absent。



请解释这个模型为什么是正确的。

2. 构造一个图 8-1 中模型的变体，使得 *clean* 和 *noisy* 状态共享同一个细化，但是行为一样。
3. 这个问题探究了共同使用 SDF 模型和模态模型来提高表达能力。实际上，除了 SDF 外，还可以使用不带指示器的简单代码来实现，用状态细化扩展模态模型。特别地，给定一个输入序列，如

(1, 1, 2, 3, 3, 3, 3, 4, 4, 4)

需要显示序列对 (i, n) ，这里 i 是输入序列中的一个数字， n 是数字重复的次数。对于上边的序列，输出应该是

$((1, 2), (2, 1), (3, 4), (4, 3))$ 。

确保解决方法符合 SDF 语义。不要使用 3.2.3 节中的非 SDF 方法。注意，这个样例可能会出现许多编码应用中，包括图像和视频编码。

连续时间模型

Janette Cardoso、Edward A. Lee、Jie Liu 和 Haiyang Zheng

连续时间模型是由 Ptolemy II 连续时间（Continuous-Time, CT）域（也称连续域）实现的，该连续域主要对物理过程进行建模。对于 CPS 系统（Cyber-Physical System, CPS）特别有用，其特征是将计算和物理过程进行融合。

CT 域从概念上将时间建模为一个连续体。它利用 Ptolemy II 中的超密时间模型处理不连续的信号、离散信号与连续信号混合的信号以及纯粹的离散信号。由此产生的模型可以分层地（hierarchically）与离散事件（discrete event）模型相结合，模态模型（modal models）可以用来开发混合系统（hybrid system）。

9.1 常微分方程

利用常微分方程（Ordinary Differential Equation, ODE）表示物理过程中的连续动态行为，主要通过带时间变量的微分方程表示。连续时间系统的使用在 Ptolemy II 模型中与 Simulink（来源于 MathWorks）中相似，但 Ptolemy 使用的超密时间使得混合信号和混合系统的模型更加清晰（Lee and Zheng, 2007）。本节的重点在于如何通过 Ptolemy II 模型对连续动态行为进行描述及连续指示器控制该模型执行。

9.1.1 积分器

在 Ptolemy II 中，反馈回路中使用 Integrator 角色来表示微分方程。在时间 t ，Integrator 角色的输出由下式给出：

$$x(t) = x_0 + \int_{t_0}^t \dot{x}(\tau) d\tau \quad (9-1)$$

式中， x_0 是 Integrator 角色的 initialState（初始状态）， t_0 是指示器的 startTime（开始时间）， \dot{x} 是 Integrator 角色的输入信号。注意，由于 Integrator 角色的输出 x 是输入 \dot{x} 的积分，所以在任何给定时间，输入 \dot{x} 是输出 x 的导数，

$$\dot{x}(t) = \frac{d}{dt} x(t) \quad (9-2)$$

因此，系统描述取决于用户对积分方程或微分方程两种形式的选择。Integrator 角色可以表示 ODE，如下例所述。

例 9.1 著名的洛伦茨吸引子（或 Lorenz 吸引子）是一个非线性反馈系统，它解释了一种称为奇异吸引子（strange attractor）的混沌行为方式。图 9-1 中的模型是控制该系统行为的非线性 ODE 的框图。积分器 1 的输出为 x_1 ，积分器 2 的输出是 x_2 ，积分器 3 的输出是 x_3 。

图 9-1 描述的方程是：

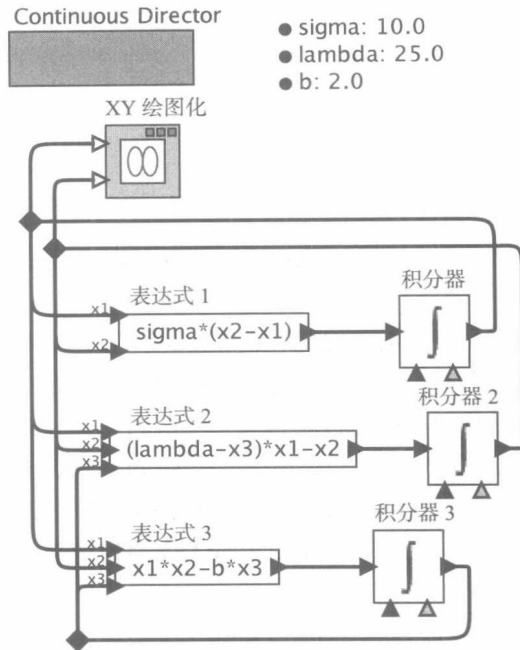


图 9-1 一组非线性常微分方程模型

$$\begin{aligned}
 \dot{x}_1(t) &= \sigma(x_2(t) - x_1(t)) \\
 \dot{x}_2(t) &= (\lambda - x_3(t))x_1(t) - x_2(t) \\
 \dot{x}_3(t) &= x_1(t)x_2(t) - bx_3(t)
 \end{aligned} \tag{9-3}$$

其中 σ 、 λ 和 b 是实数常数。对于每个方程，模型中等号右边的表达式由一个 Expression 角色实现，它的图标名称为 Expression。每个表达式中的参数（如 λ 为 λ ， σ 为 σ ）和角色的输入端口（如 x_1 为 x_1 ， x_2 为 x_2 ）已经在方程中指定。每个 Expression 角色中的表达式可以通过双击角色进行编辑；参数值可以通过双击参数进行编辑，参数显示图上方。

3 个积分器指定 x_1 、 x_2 和 x_3 的初始值，这些值可以通过双击相应的 Integrator 图标来修改。在该例中，所有的 3 个初始值设置为 1.0（图中没有显示）。

显示在图中左上方的 Continuous Director 管理模型的仿真。它包含一个带有多个关键参数的复杂 ODE 求解器。可以通过双击该指示器在图 9-2 所示对话框访问这些参数。

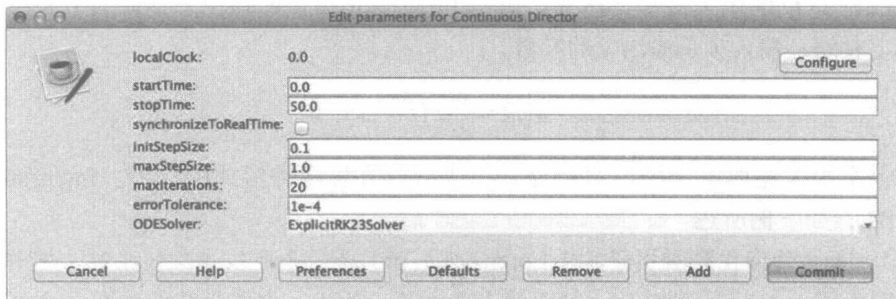


图 9-2 显示图 9-1 中模型的指示器参数的对话框

最简单的参数是 `startTime` 和 `stopTime`，它们定义仿真将要执行的时间轴的范围。其他

参数的作用将在练习 1 中进行探讨。

Lorenz 模型的输出如图 9-3 所示。对于 startTime 和 stopTime 之间的 t 值, XY 绘图仪图示了 $x_1(t)$ 对 $x_2(t)$ 的奇异吸引子图。

与 Lorenz 模型一样, 许多连续时间模型在反馈循环回路中包含积分器。接下来将介绍如何使用更复杂的模块而不仅使用 Integrator 角色来实现线性和非线性动态过程。

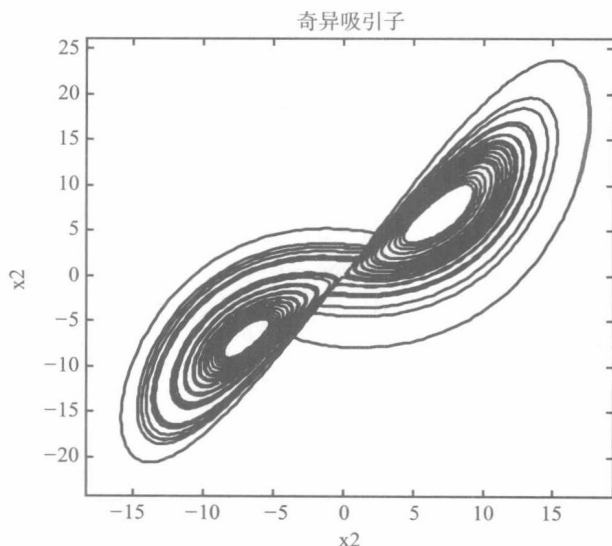


图 9-3 Lorenz 模型运行的结果

9.1.2 传递函数

在描述连续时间系统时, 通常使用高层次的描述比单个积分器更方便。例如, 对于线性时不变 (Linear Time Invariant, LTI) 系统, 它通常用传递函数 (transfer function) 来描述输入输出行为, 它是对脉冲响应的拉普拉斯变换 (Laplace transform)。具体来说, 对于输入 x 和输出 y , 传递函数可以用一个关于复变量 s 的函数来给出:

$$H(s) = \frac{Y(s)}{X(s)} = \frac{b_1 s^{m-1} + b_2 s^{m-2} + \dots + b_m}{a_1 s^{n-1} + a_2 s^{n-2} + \dots + a_n} \quad (9-4)$$

Y 和 X 分别是 y 和 x 的拉普拉斯变换。分母系数的个数 n 严格大于分子系数的个数 m 。传递函数描述的系统可通过单个积分器来构建, 但在 Ptolemy 系统中使用 ContinuousTransferFunction (连续传递函数) 角色更方便实现, 如下所述。

例 9.2 考虑在图 9-4 中的模型, 它产生图 9-5 中的图形。这个模型使用 ContinuousClock(连续时钟)角色 (见第 9 章补充阅读: 连续时间信号发生器) 生成一个方波, 然后将方波提供给 ContinuousTransferFunction 角色。ContinuousTransferFunction 角色实现的传递函数如下所示:

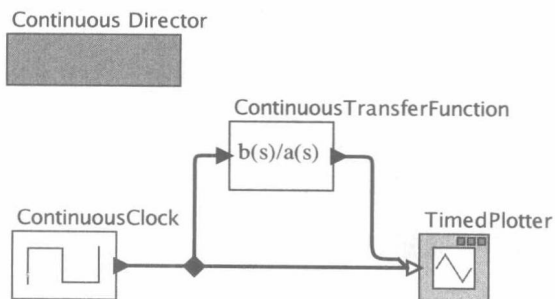


图 9-4 一个 ContinuousTransferFunction 使用范例

$$H(s) = \frac{Y(s)}{X(s)} = \frac{1}{0.001s^2 + 0.01s + 1}$$

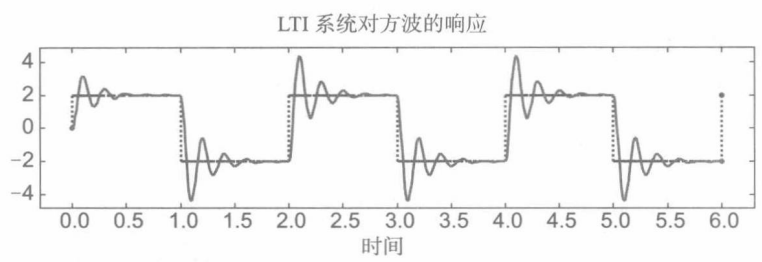


图 9-5 图 9-4 中模型的运行结果

比较式 (9-4), 可以看到, $m = 1$ 和 $n = 3$, 其他参数如下:

$$b_1 = 1$$

$$a_1 = 0.001, a_2 = 0.01, a_3 = 1.0$$

因此角色的参数设置为

numerator= {1.0}

denominator = {0.001, 0.01, 1.0}

用单个积分器组成的等效模型如图 9-6 所示 (练习 2 将探索为什么这些是等效的)。

前面的例子表明, 由积分器、放大器和加法器构成的复杂网络可以使用 Continuous TransferFunction 角色来简洁地表示。事实上, 这个角色使用指定的参数值来构造类似于图 9-6 所示的层次模型。通过右击 ContinuousTransferFunction 角色并选择 Open Actor 来查看这个层次模型 (选择 [Graph→Automatic Layout], 这样所有角色能够在更具可读性的布局中显示)。ContinuousTransferFunction 是一个高阶角色的例子, 其中参数指定了实现角色功能的角色网络。

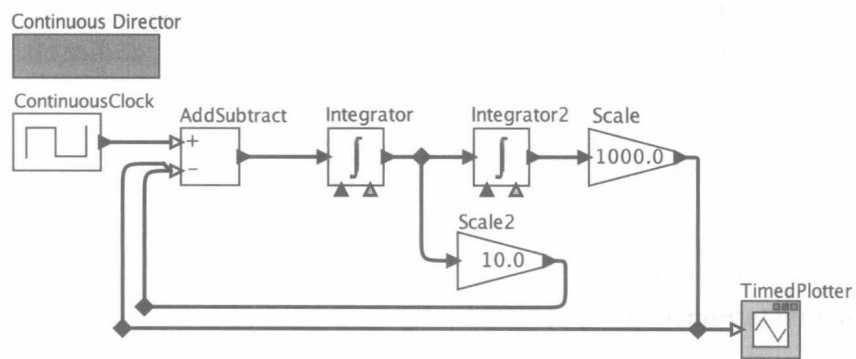


图 9-6 相当于图 9-4 中模型的一个模型, 假设例 9.2 中的参数

ContinuousTransferFunction 角色和支持动态高级描述的其他角色详见第 7 章补充阅读: 用于动态建模的角色。

9.1.3 求解器

数值积分是一个古老的、复杂的和深奥的主题 (Press et al., 1992)。这个主题的完整描述超出了本书的范围, 但是为了有效地利用 Ptolemy 求解器函数 (使用数值积分来求方程的

解), 理解这些基本概念是非常有用的。本节将简要概述在 Ptolemy II Continuous 指示器实现的求解器机制。

假设 w 是一个连续时间信号。暂时先忽略 Ptolemy II 中使用超密时间模型的情况, 并假设 w 是一个可积分的函数, 其形式为 $w: \mathbb{R} \rightarrow \mathbb{R}$ 。进一步假设, 对于任何 $t \in \mathbb{R}$, 都可以得到一个 $w(t)$ 。再假设 x 是一个由下式给出的连续时间信号

$$x(t) = x_0 + \int_0^t w(\tau) d\tau$$

其中 x_0 是一个常数。 $x(t)$ 等价于曲线 $w(\tau)$ 从 $\tau=0$ 到 $\tau=t$ 的面积, 加上一个初始值 x_0 。注意, w 是 x 的导数, 或表示为 $w(t)=\dot{x}(t)$ 。给定 w , 可以通过把 w 作为输入, 然后输入到一组带有 initialState (初始状态) 集 x_0 的 Integrator 角色中来构造 x ; 然后输出 x 。

数值积分 (numerical integration) 是使用足够多的 $t \in \mathbb{R}$ 的点来准确推断函数形状的过程。当然, “准确的” 意义可能取决于具体应用, 但有一个关键的标准是 x 的值在充分多的点足够准确, 这些 x 值可以用来计算在其他时间点 $t \in \mathbb{R}$ 的 x 值。求解器 (solver) 是数值积分算法的一个实现。最简单的求解器是固定步长求解器 (fixed step size solver)。该求解器首先定义一个步长 (step size) h , 计算以 h 为间隔的 x 值, 特别是 $x(h)$ 、 $x(2h)$ 、 $x(3h)$ 等。

合理准确的固定步长求解器采用梯度法 (trapezoidal method), 其中 x 由下式近似给出:

$$x(nh) = \begin{cases} x_0 & n = 0 \\ x[(n-1)h] + h[w((n-1)h) + w(nh)]/2 & n \geq 1 \end{cases}$$

该方法如图 9-7 所示。从上式可知曲线 w 从 $0 \sim nh$ 的面积由宽度为 h 的梯形区域的面积和来近似, 其中梯形两边的高度由不同 $w(mh)$ 给出 (m 是整数)。图中的阴影梯形的面积近似为曲线从时间 $(n-1)h$ 到 nh 下的面积, 其中 $n=6$ 和 $h=0.05$ 。

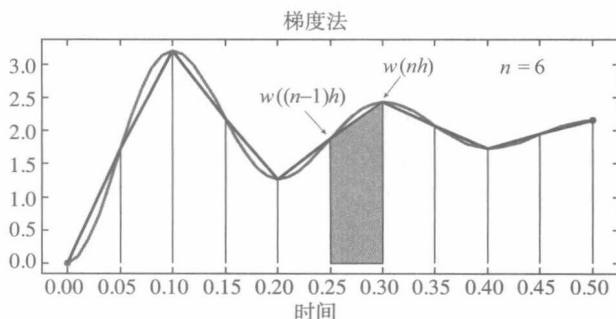


图 9-7 梯度法的说明图。曲线下的面积用梯形的面积和来近似。一个梯形如上图阴影区

然而, 梯形法求解器在反馈系统中很难使用, 如图 9-1 所示, 因为求解器需要对基于输入的 Integrator (积分器) 角色的输出结果进行计算。为了在时间 $t_n = nh$ 时计算一个 Integrator (积分器) 的输出, 求解器需要知道在 $t_{n-1} = (n-1)h$ 和 $t_n = nh$ 时的输入值。但在图 9-1 中, 在任意时 t_n 积分器的输入取决于在同一时刻 t_n 相同的积分器的输出, 有一个循环依赖。表现出循环依赖的求解器称为隐式 (implicit method) 求解器。反馈系统中使用它们的一种方法是 “猜” 反馈值, 然后反复地修改猜测值直到达到所需要的精度, 但通常这些策略能否产生最优解是没有任何保证的。

同隐式求解器相比, 前向欧拉法 (forward Euler method) 是一种显式法求解器 (explicit method)。它类似于梯度法, 但它更容易应用于反馈系统。通过以下公式近似得到 x 。

$$x(nh) = x[(n-1)h] + hw[(n-1)h]$$

该方法如图 9-8 所示。每一步长的面积近似为矩形，而不是梯形。这种方法不太精确，通常，误差积累更快，但它不要求解器在时间 nh 知道输入的情况。

一般来说，使用一个更小的步长 h 可以增加解的准确性，但同时也增加了计算量。满足准确性目标所需要的步长取决于信号的变化速度。梯度法和前向欧拉法可以推广为可变步长（variable step size）求解器，它基于信号的变化动态调整步长。求解器在时刻 t_1 、 t_2 等处评估积分，通过算法在时间增量下立即求出积分增量。该算法首先选择一个步长，执行数值积分，然后估计误差。如果误差的估计高于某个阈值（由指示器的 `errorTolerance` 参数控制），那么指示器用一个更较小的步长来重新执行数值积分。

可变步长前向欧拉求解器首先确定一个时间增量 h_n 来定义 $t_n = t_{n-1} + h_n$ ，然后计算

$$x(t_n) = x(t_{n-1}) + h_n w(t_{n-1})$$

可变步长前向欧拉法是被广泛使用的 Runge-Kutta (RK) 方法的一个特例。连续指示器 (Continuous director) 提供了 RK 求解器的两个变体，`ExplicitRK23Solver` 和 `ExplicitRK45Solver`，选择使用指示器的 `ODESolver` 参数。关于这些变体更详细描述见第 7 章补充阅读：Runge-Kutta 方法。图 9-5 中的图是使用 `ExplicitRK23Solver` 生成的。逼近曲线的截面表明求解器选择计算信号值的时刻，如图 9-9 所示。该图表明，求解器在信号变化较快的区域使用较小的步长。

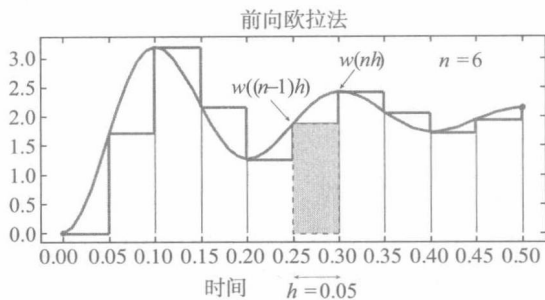


图 9-8 前向欧拉法的说明。曲线下的面积用矩形的面积和来近似，如上图阴影部分

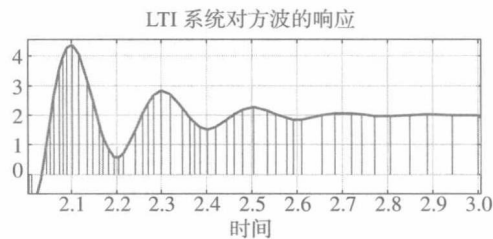


图 9-9 图 9-5 中图的逼近曲线（用截面显示）表明，求解器在信号变化更快的区域使用更小的步长

补充阅读：连续时间信号发生器

连续域提供了多个生成连续时间信号的角色。



这些角色可在 `DomainSpecific` → `Continuous` → `SignalGenerators` 中找到，`BandlimitedNoise` 除外，它在 `DomainSpecific` → `Continuous` → `Random` 中。

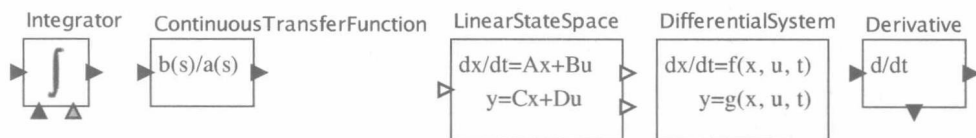
- `ContinuousClock` 的参数与 `DiscreteClock` 类似，但它产生分段常数（piecewise constant）信号。如图 9-5 所示的方波就是这样信号的一个简单例子，这个角色也

能产生复杂的重复或不重复形式。

- **ContinuousSinewave**, 顾名思义, 产生一个正弦波。正弦波的频率、相位和振幅通过参数设置。这个角色约束求解器的步长, 以确保它的输出是平滑的。步长设置为不大于正弦波周期 $1/10$ 的值。
- **BandlimitedNoise** 是最复杂的信号发生器。这个角色产生服从高斯分布且具有可控制带宽的连续时间噪声。虽然该话题的完整讨论超出了本书的范围, 但值得注意的是, 从理论上来说一个因果系统产生理想的限带噪声是不可能的 (见 Lee and Varaiya (2011))。这个角色实现了合理的近似值。与 **ContinuousSinewave** 类似, 这个角色影响求解器的步长选择, 它确保求解器至少以指定带宽两倍的频率采样信号。这是理想的限带噪声信号的奈奎斯特频率。
- **Waveform** 从一个有限的样本集中产生周期性的波形。它提供了两种插值方法, 线性插值和 Hermite 插值, 后者使用一个基于第 11 章中 Foley et al. (1996) 提出的 Hermite 曲线的三阶插值方法。Hermite 插值对于生成任意形状的光滑曲线是很有用的。这种插值方法假设波形是周期性的。注意, 这个角色也会影响求解器所采用的步长。特别是, 它确保求解器包括指定的样本, 虽然它不要求求解器包括它们之间的任何样本。

补充阅读：用于动态建模的角色

Ptolemy II 提供多个角色, 它们可用于对具有复杂动态 (行为随着时间的推移变化) 的连续时间系统建模的角色。这些角色如下所示, 可在 `DomainSpecific` → `Continuous` → `Dynamics` 中找到。



基础角色是 9.1.1 节和 9.2.4 节描述的 Integrator (积分器角色)。其他的是使用 Integrator 实例中的子模型构成的高阶角色。

如 9.1.2 节解释的, **ContinuousTransferFunction** 可以实现基于指定为两个多项式比的传递函数的连续时间系统。**ContinuousTransferFunction** 角色不支持 Integrator 的非零初始条件。

- **LinearStateSpace** 指定带有一组描述线性常系数差分方程 (LCCDE) 矩阵和向量系统的输入输出关系。与 **ContinuousTransferFunction** 角色不同, 这个角色支持非零初始条件, 但它同样受到与可以用线性函数描述模型系统同样的限制。
- **DifferentialSystem** 可用于对复杂的非线性动态建模。例如, 它可以用来指定例 9.1 中的 Lorenz 吸引子, 如练习 3 所述。有关详细信息请参阅角色说明文档。
- **Derivative** 提供了其输入求导的粗略估计。使用这个角色是不乐观的, 因为即使输入是连续和可微的, 但是它的输出可能有噪声。输出仅仅是当前输入与前面输入的差除以步长的值。然而, 如果输入是不可微的, 那么输出就不是分段连续的, 这些输出可能迫使求解器使用允许的最小步长。注意, 该角色有两个输出。

当输入不连续时，下部输出产生一个离散事件。该事件表示一个狄拉克 δ 函数 (Dirac delta function)，如 9.2.4 节所述)。

补充阅读：Runge-Kutta (龙格-库塔) 方法

一般来说，ODE 可以用关于矢量值状态的微分方程组来表示

$$\dot{x}(t) = g(x(t), u(t), t)$$

$$y(t) = f(x(t), u(t), t)$$

其中， $x: \mathbb{R} \rightarrow \mathbb{R}^n$ ， $y: \mathbb{R} \rightarrow \mathbb{R}^m$ 和 $u: \mathbb{R} \rightarrow \mathbb{R}^l$ 分别是状态、输出和输入信号。函数 $g: \mathbb{R}^n \times \mathbb{R}^l \times \mathbb{R} \rightarrow \mathbb{R}^n$ 和 $f: \mathbb{R}^n \times \mathbb{R}^l \times \mathbb{R} \rightarrow \mathbb{R}^m$ 分别是状态函数和输出函数。状态函数 g 由图 9-1 的反馈路径中的 Expression 角色表示。这个函数表明了 Integrator 角色的输入 $x(t)$ 是它们输出 $x(t)$ 、外部输入 $u(t)$ (图 9-1 中没有) 和当前时间 t (图 9-1 也没有使用) 的函数。

给出了这个公式，一个显式的 k 阶 RK 方法表示如下

$$x(t_n) = x(t_{n-1}) + \sum_{i=0}^{k-1} c_i K_i \quad (9-5)$$

其中

$$K_0 = h_n g[x(t_{n-1}), u(t_{n-1}), t_{n-1}]$$

$$K_i = h_n g[x(t_{n-1}) + \sum_{j=0}^{i-1} A_{i,j} K_j, u(t_{n-1} + h b_i), t_{n-1} + h b_i] \quad i \in \{1, \dots, k-1\}$$

还有， $A_{i,j}$ 、 b_i 和 c_i 是通过比较 x 的泰勒级数展开式 (式 (9-5)) 计算而得到的算法参数。

一阶 RK 方法，也称为前向欧拉方法，有 (简单得多)，如下形式：

$$x(t_n) = x(t_{n-1}) + h_n \dot{x}(t_{n-1})$$

该方法在概念上非常重要，但没有其他可用方法准确。

更精确的 Runge-Kutta 方法有三阶或四阶，并控制每个积分步的步长。由 Continuous 指示器实现的 ExplicitRK23Solver 是 $k=3$ (三阶) 方法，由下式给出

$$x(t_n) = x(t_{n-1}) + \frac{2}{9} K_0 + \frac{3}{9} K_1 + \frac{4}{9} K_2 \quad (9-6)$$

其中

$$K_0 = h_n g[x(t_{n-1}), t_{n-1}] \quad (9-7)$$

$$K_1 = h_n g[x(t_{n-1}) + 0.5 K_0, u(t_{n-1} + 0.5 h_n), t_{n-1} + 0.5 h_n] \quad (9-8)$$

$$K_2 = h_n g[x(t_{n-1}) + 0.75 K_1, u(t_{n-1} + 0.75 h_n), t_{n-1} + 0.75 h_n] \quad (9-9)$$

注意，为了完成一个积分步，这种方法需要评估函数 g 除了时间 t_{n-1} 之外， t_{n-1} 到 t_n 时间之间在 $t_{n-1} + 0.5 h_n$ 和 $t_{n-1} + 0.75 h_n$ 中间时间的值，其中 h_n 表示步长。这一事实大大复杂了角色的设计，因为它们必须容忍多个推测由你的评估值 (点火)，如果所需的精度没有达到，可能不得不以一个更小的步长重做。直到全部积分步前完成步长 h_n 的有效性都是未知的。事实上，任何需要对状态函数 g 的值进行瞬时评估值的方法都会遇到同样的问题，如经典的四阶 RK 方法、线性多步方法 (LMS) 和 BulirschStoer 方法。

在连续域中，RK 求解器在中间点上推测执行模型，调用角色的 fire 方法，但不是

它们的 `postfire` 方法。因此，在连续域中使用的角色都必须符合严格角色语义，它们不能改变 `fire` 方法中的状态。

9.2 离散和连续的混合系统

连续域支持离散和连续行为的混合。最简单的这种混合产生分段连续信号，除了在特定的时间点发生突然改变外，它们随时间平滑变化。分段连续信号将在 9.2.1 节中详细描述。

此外，信号可以是真正的离散。比如，与 DE 域一样，连续域中的信号可以在某个时间戳缺失。从来没有缺失的信号是真正的连续时间信号 (continuous-time signal)。除了在一组离散时间戳集合外，总是缺失的信号是离散事件信号 (discrete-event signal)，见 9.2.2 节。混合两种类型信号的模型是混合信号模型 (mixed signal model)。它可以在一个时间戳范围内有连续时间信号，另一个范围内有离散事件，这些称为混合信号 (mixed signal)。

9.2.1 分段连续信号

如图 9-9 所示，当信号快速变化时，可变步长求解器每单位时间将产生更多的样本。然而，这些求解器不能直接支持不连续信号，如图 9-5 所示的方波。Ptolemy II 中的连续指示器增加了可处理这种不连续情况的 ODE 求解器，但信号必须是分段连续的。满足这个先决条件需要注意一些情况，这将在本章后面讨论。

前面讲过，Ptolemy II 使用超密时间模型。这意味着连续时间信号的函数形如：

$$x: T \times \mathbb{N} \rightarrow V \quad (9-10)$$

式中， T 是模型时间值的集合 (见 1.7.3 节)， \mathbb{N} 是表示微步的非负整数， V 是值的集合 (集合 V 是信号的数据类型)。该函数指定，在每个模型时间 $t \in T$ ，信号 x 可以有多个值，这些值按照指定的顺序出现。例如，对于图 9-5 中的方波，在时间 $t = 1.0$ ，方波的值首先是 $x(t, 0) = 2$ ，然后是 $x(t, 1) = -2$ 。

为了让时间超过模型时间 $t \in T$ ，需要确保模型中的每个信号在时间 t 有数量有限的值。因此，要求对于所有 $t \in T$ ，存在一个 $m \in \mathbb{N}$ 使得

$$\forall n > m, x(t, n) = x(t, m) \quad (9-11)$$

这个约束可以防止在特定时间内信号呈现出无穷多值的抖动的芝诺条件。这些条件防止在模型时间点之外的进程执行，假设约束执行产生按时间顺序的值。

假设 x 不满足抖动的芝诺条件，那么就至少有一个 m 值满足式 (9-11)。 m 的值称为终止微步 (final microstep)， $x(t, m)$ 为 x 在时间 t 的终值 (final value)。 $x(t, 0)$ 称为时间 t 的初值 (initial value)。如果 $m = 0$ ，那么可以说 x 在时间 t 只有一个值。

通过下式定义初值函数 $x_i: T \rightarrow V$

$$\forall t \in T, x_i(t) = x(t, 0)$$

通过下式定义终值函数 $x_f: T \rightarrow V$

$$\forall t \in T, x_f(t) = x(t, m_t)$$

其中 m_t 是在时间 t 的终止微步。注意，如果将模型时间抽象为实数 $T = \mathbb{R}$ ，那么 x_i 和 x_f 是传统的连续时间函数。

分段连续信号可以定义为形如 $x: T \times \mathbb{N} \rightarrow V$ 的函数 x ，它不满足抖动芝诺条件的 3 个要求：

- 1) 初值函数 x_i 是左连续的;
- 2) 终值函数 x_f 是右连续的;
- 3) x 在所有 $t \in T \setminus D$ 中只有一个值, 其中 D 是 T 的一个离散子集。

最后一个要求很微妙, 值得进一步讨论。首先, $T \setminus D$ 是指包含集合 T 所有元素的集合, 除了在集合 D 中的元素。 D 限制为一组离散集 (见第 9 章补充阅读: 进一步探索离散集)。直观来看, D 是一组可以按照时间顺序计算的时间值的集合。很容易看到, 如果 $D = \emptyset$ (空集), 那么 $x_i = x_f$, x_i 和 x_f 都是连续函数。否则这些函数是分段连续的。

按照上述说明, 在 Ptolemy II 中连续域的关键约束是所有信号是分段连续的。基于这个定义, 图 9-5 中的方波是分段连续的。在每个断点, 它有两个值: 一个与该断点前的值匹配的初值; 一个与该断点后的值相匹配的终值。创建不分段连续的信号比较简单, 如下例所示。

例 9.3 图 9-10 中的模型包含一个输入是连续时间信号的 Expression 角色。表达式如下所示:

$(in > 1.0) ? in + 1 : 0$

如果输入大于 1.0, 则输出将等于输入加 1; 否则, 输出将是 0。按照上面的描述, 这个输出信号不是分段连续的。在时间 1.0 或之前, 输出值为 0。但在 1.0 之后的任何时间, 值不为 0。这个信号向右是不连续的。

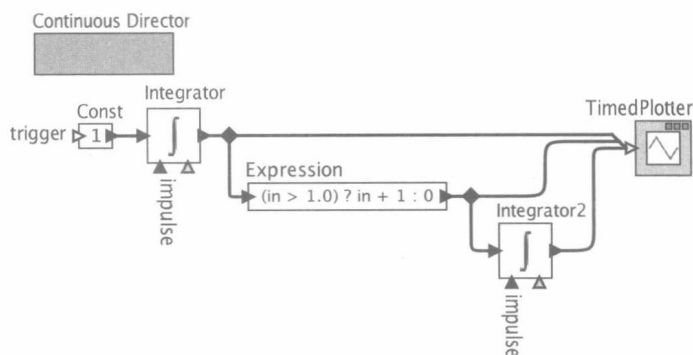


图 9-11 显示了产生的结果 图 9-10 一个模型, 它产生的信号不是分段连续的, 因此可以展示求解器行为。图 9-23 中的模型解决了这个问题

图, 其中 Expression 角色的输出标记为 “second”。从 0 到非 0 的转移不是瞬时的, 如中间图的斜虚线所示。更糟糕的是, 转移的宽度取决于模型看似无关紧要的细节。模型说明了连接到第二个积分器的信号。如果将第二积分器从模型中删除, 那么转移的宽度改变, 如底图所示。这是因为第二积分器影响求解器所采用的步长。为了达到合适的积分精度, 第二积分器迫使在时间 1.0 附近有一个更小的步长。

该问题没有通过改变下式而得到解决

$(in \geq 1.0) ? in + 1 : 0$

这里, 当输入大于或等于 1.0 时, 将产生从 0 到非 0 的转移。在这种情况下, 产生的信号向左不连续。由此产生的图都是相同的。这个 Expression 角色在点火时简单计算其输入的特定函数。它没有在不同的微步产生不同值的机制, 除非其输入已经在不同微步有不同值。

前面的例子表明, 如果输入是连续时间信号, 那么使用输出的不连续信号函数的角色将会产生问题。接下来的几节将描述各种正确构造不连续信号的机制。图 9-10 中的特定问题将在 9.3.1 节使用模态模型解决。

补充阅读: 进一步探索离散集

如果 D 是完全有序集 (对于任何两个元素 d_1 和 d_2 , 有 $d_1 \leq d_2$ 或 $d_1 > d_2$), 存在一个

一对一的保存 (order preserving) 函数 $f: D \rightarrow \mathbb{N}$, 那么 D 是一个离散集 (discrete set)。保序方法仅仅意味着, 对于所有的 $d_1, d_2 \in D$, 当 $d_1 \leq d_2$ 时, 有 $f(d_1) \leq f(d_2)$ 。这种一对一函数的存在保证 D 的元素可以按时排序。注意, D 是一个可数集, 但并不是所有的可数集都是离散的。例如, 有理数集 \mathbb{Q} 可数但不离散。但不存在这种一对一的函数。

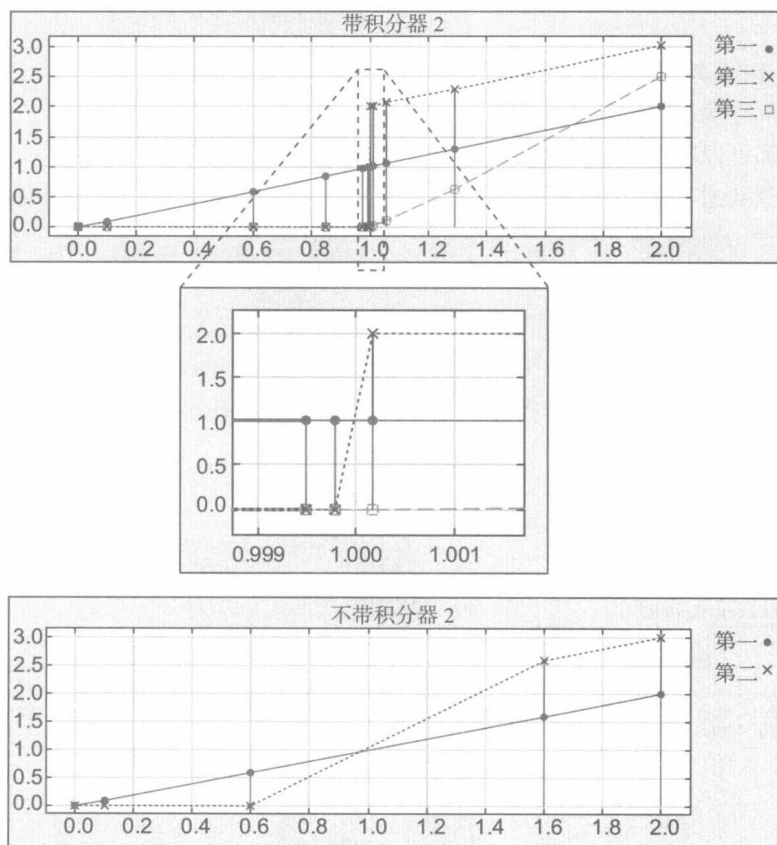


图 9-11 在有与没有第二个积分器时图 9-10 中的模型产生的信号。注意, Expression (表达式) 角色的输出似乎取决于第二积分器是否存在。信号 “first” “second” 和 “third” 分别是图中从上到下的输入

9.2.2 连续域中的离散事件信号

如前文所述, 连续域支持在特定时刻出现的真正离散信号。因此, DE 域中使用的时钟角色 (见第 7 章补充阅读: 时钟角色) 可用于连续域。

例 9.4 图 9-4 中的 ContinuousClock 角色是一个使用 DiscreteClock 和 ZeroOrderHold (见第 7 章补充阅读: 离散事件中的连续信号) 的复合角色, 如图 9-12 所示。DiscreteClock 产生离散事件信号, ZeroOrderHold 将信号转换为连续时间信号 (见第 7 章补充阅读: 离散事件中的连续信号)。

这两个信号都是分段连续的。在每个事件的模型时间，DiscreteClock 的输出信号描述如下：在微步 0 时（它匹配该事件前时间的值），它为缺失；在微步为 1 时（它是离散事件），它为存在；在微步为 2 或更高时（它匹配更大时间上它的值直到下一个离散事件），它再次为缺失。因此，根据求解器的要求，DiscreteClock 的输出是分段连续的。

第 7 章补充阅读：时钟角色描述的时钟角色的所有行为都与 DiscreteClock 类似，因此它们都可以在连续域中使用。

注意，尽管操作或产生离散事件的许多角色都有一个 trigger 输入端口，但是在连续域中很少连接这个端口。在 DE 域中，trigger 端口用于在输入事件的时间触发角色的执行。但在连续域中，每个角色每时每刻执行。尽管如此，有时使用 trigger 端口是有用的，如图 9-13 所示。

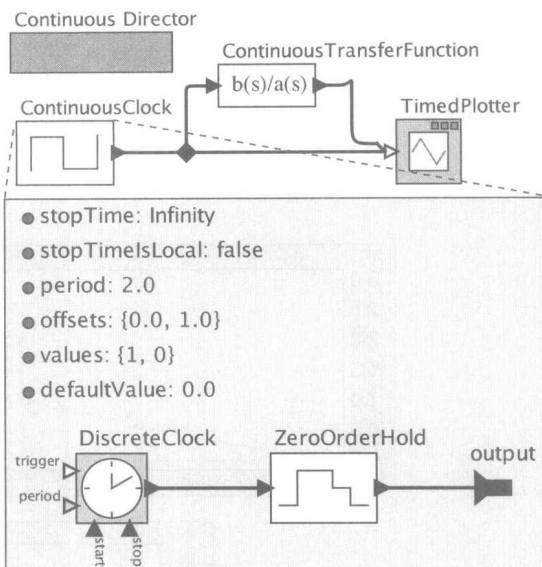


图 9-12 模型中的 ContinuousClock 角色是一个使用 DiscreteClock 和 ZeroOrderHold 的复合角色

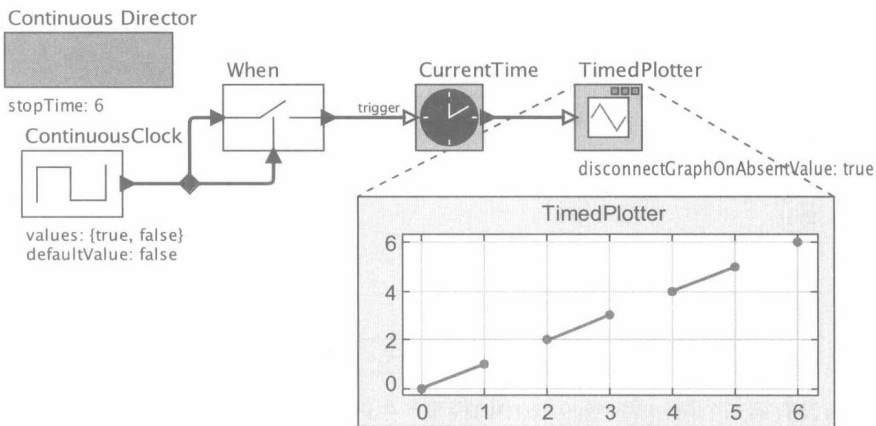


图 9-13 模型中 CurrentTime 角色的 trigger 端口用于打开和关闭它的输出。在 DiscreteClock 角色为 false 的时间间隔内，CurrentTime 角色是无效的，因此它的输出是缺失

9.2.3 离散时间的积分器重置

除了信号输入和输出端口外，Integrator 角色在图标下部有两个额外的端口。右下方是一个叫作 initialState 的端口参数。当在那个端口上提供一个输入令牌时，Integrator 的状态将重置为令牌的值。Integrator 的输出将立即更改为指定的值。

例 9.5 图 9-14 中的模型使用 DiscreteClock 角色定期重置 Integrator。

要求 initialState 端口上的输入事件是纯离散的。这意味着，在所有的模型时间，输入信号必须在微步 0 时为缺失。任何将连续信号送入这个端口的尝试都导致类似如下的异常：

`IllegalActionException: Signal at the initialState port is not purely discrete.`

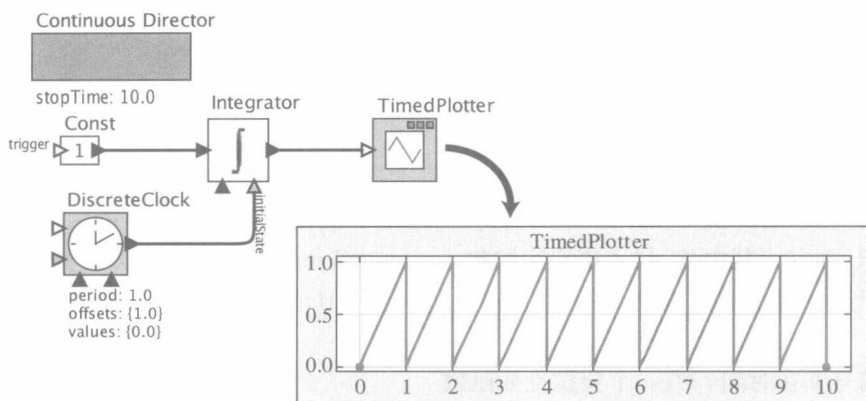


图 9-14 Integrator (积分器) 角色的 initialState 端口的使用说明

in Integrator

这个检查确保积分器的输出是分段连续的。在图 9-14 中, 例如, 在不连续的点, 信号首先取应用重置前的 Integrator 状态的值。在随后的微步中, 它取重置后的值。相反, 如果 Integrator 的输出在微步 0 时突然改变, 那么在以前的无穷多个时间, 输出可能是不同的值, 因此违反了分段连续性的要求。

9.2.4 狄拉克 δ 函数

另一个积分器下部输入端称为脉冲 (impulse)。与前一节所讨论的 initialState 端口一样, 这个端口的信号需要是纯离散的。当脉冲事件到达时, 它导致积分器状态 (和输出) 的瞬时递增或递减。也就是说, 不是将状态重置为一个指定的值, 而是在当前状态的基础上增加或减少。

补充阅读: 离散事件中的连续信号

如下图所示, ZeroOrderHold 角色需要一个离散事件信号输入, 并在它的输出产生一个连续时间信号:

这个角色在 DomainSpecific \rightarrow Continuous \rightarrow Discrete to Continuous 可找到。

在输入事件之间的时间, 输出值是最近事件的值, 所以输出是分段常数。在每个输入事件时间, 在微步 0 的输出是前面事件的值; 在微步 1, 它取当前事件的值。因此, 输出信号是分段连续的。

定义一个角色, 使它可以在输入事件的值之间插值, 就像 Waveform 角色所做的一样 (见第 9 章补充阅读: 连续时间信号发生器) 是很有必要的。然而, 为了插入值, 角色需要知道未来事件的值。然而在连续域中的角色要求是存在因果关系的, 这意味着它们的输出只取决于当前和过去的输入。输出不能依赖于未来的输入。因此, 这样的插值是不可能的。Waveform 角色能够进行插值, 因为它的插值通过参数指定, 而不是通过输入事件来确定。



数学上, 在信号与系统中, 这种功能通常表示为狄拉克 δ 函数 (Dirac delta function)。狄拉克 δ 函数是一个由下式给出的函数 $\delta: \mathbb{R} \rightarrow \mathbb{R}^+$

$$\forall t \in \mathbb{R}, t \neq 0, \delta(t) = 0$$

和

$$\int_{-\infty}^{\infty} \delta(\tau) d\tau = 1$$

也就是说,除了在 $t = 0$ 时刻外,信号值都为零,而在整个定义域上的积分却为 1。因此,在 $t = 0$ 时,它的值不具体定义。只限定取值为零的区域,而这个值由函数 $\delta: \mathbb{R} \rightarrow \mathbb{R}^+$ 的形式中的 \mathbb{R}^+ 确定,其中 \mathbb{R}^+ 代表扩充实数集 (extended real),包括无穷大。狄拉克 δ 函数被广泛应用于连续时间系统建模 (见 Lee and Varaiya (2011)), 所以能够在仿真中包括它们是重要的。

假设信号 y 在时间 t_1 有一个狄拉克 δ 函数

$$y(t) = y_1(t) + K\delta(t - t_1)$$

其中 y_1 是一个普通的连续时间信号, K 是一个比例常数。那么

$$\int_{-\infty}^t y(\tau) d\tau = \begin{cases} \int_{-\infty}^t y_1(\tau) d\tau & t < t_1 \\ K + \int_{-\infty}^t y_1(\tau) d\tau & t \geq t_1 \end{cases}$$

分量 $K\delta(t - t_1)$ 是一个在时间 t_1 有一个权重 K 的狄拉克 δ 函数,它导致在 $t = t_1$ 时积分值瞬时增加 K 。

例 9.6 线性时不变 (LTI) 系统可以用它的脉冲响应表示,其为对于狄拉克 δ 函数的响应。图 9-15 中的模型是一个带有以下传递函数的线性时不变系统。

$$H(s) = \frac{1}{1 + as^{-1} + bs^{-2}}$$

该模型在时间 0.2 时提供了一个狄拉克 δ 函数,它产生的脉冲响应如图 9-15 所示。

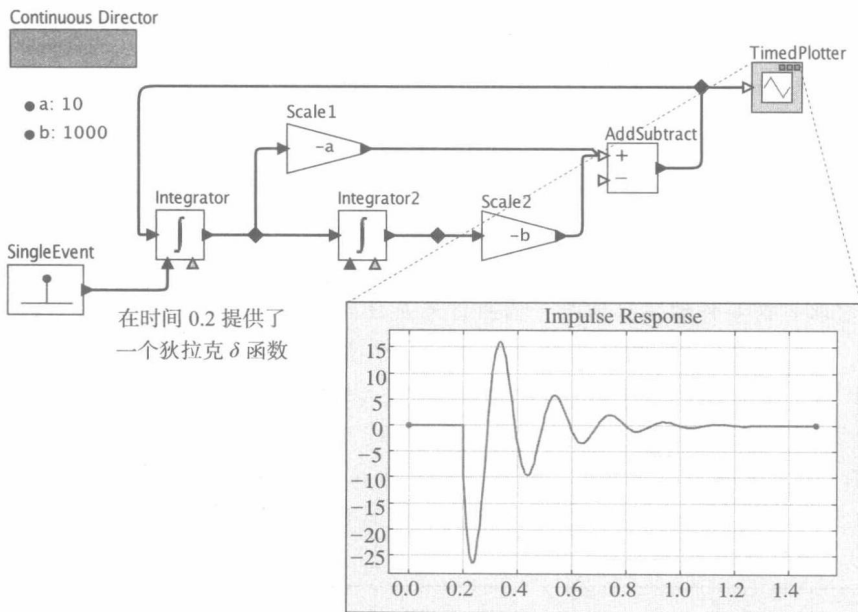
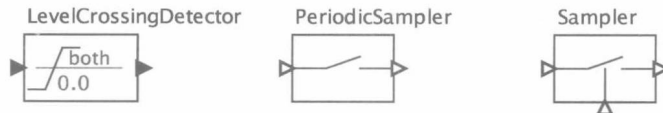


图 9-15 LTI 系统对狄拉克 δ 函数的响应

由于狄拉克 δ 函数的瞬时和无限特性，很难在计算机系统中对狄拉克 δ 函数建模。Ptolemy II 的超密时间模型与连续域的语义相结合提供了一个严格的和明确的模型，该模型可以支持狄拉克 δ 函数。

补充阅读：生成离散事件

多个角色将连续时间信号转换成离散事件信号（这些角色可在 DomainSpecific \rightarrow Continuous \rightarrow Continuous to Discrete 中找到）：



- 当输入信号经过由 level 参数指定的阈值时，LevelCrossingDetector 将连续信号转换成离散事件。direction 参数要求角色检测仅上升或者下降的转移。在这个角色产生输出前它有一个微步的延迟。也就是说，当检测到跨层时，这个角色在当前时间的下一个微步请求再点火（refiring），并在再点火期间产生输出。这将确保输出满足分段连续性约束；它在微步 0 总是缺失。一个微步的延迟使角色可以用于反馈回路。例子如图 9-16 所示，每次达到一个阈值时，反馈回路就重置积分器（例中为 1.0）。

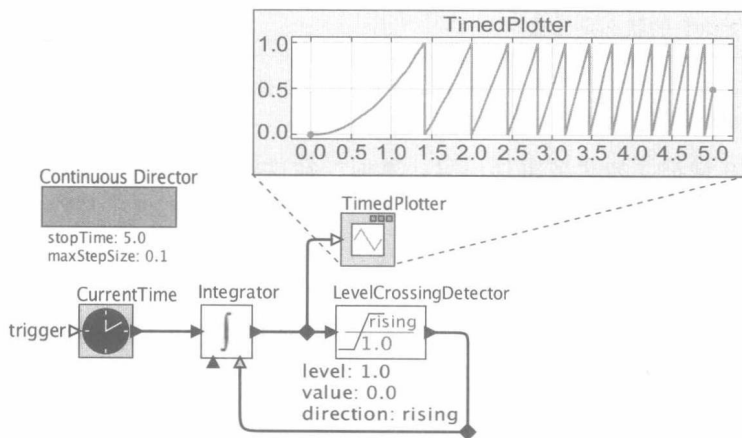


图 9-16 LevelCrossingDetector 的说明，它可以放在反馈回路中。在这种情况下，只要 Integrator 的输出达到 1.0，就重置为 0

- PeriodicSampler 通过定期采样输入信号来生成离散事件。采样率由一个参数指定。默认情况下，这个角色读取输入信号的初始值（在微步 0 的输入值），但一个微步（在微步 1）后将它发送到输出端口。这将确保在微步 0 输出总是缺失，从而确保输出信号是分段连续的（在采样时间之前输出总是缺失，所以分段连续性要求在采样时间的微步 0 它是缺失）。由于一步延迟，PeriodicSampler 也可以用于反馈回路。例如，它可以用来定期重置积分器，如图 9-17 中的例子所示。
- Sampler 是一个简单的角色。每当触发信号（图标底部端口）出现时，它将输入从左边端口复制到输出端口。没有微步延迟。如果 trigger 端口的信号是分段连续

的离散事件信号，那么输出也将是分段连续的离散事件信号。因为 trigger 输入是离散的，所以 Sampler 通常会在微步 1 读它的输入，(PeriodicSampler 会以同样的方式呈现，如果其 microstep 参数设置为 1)。

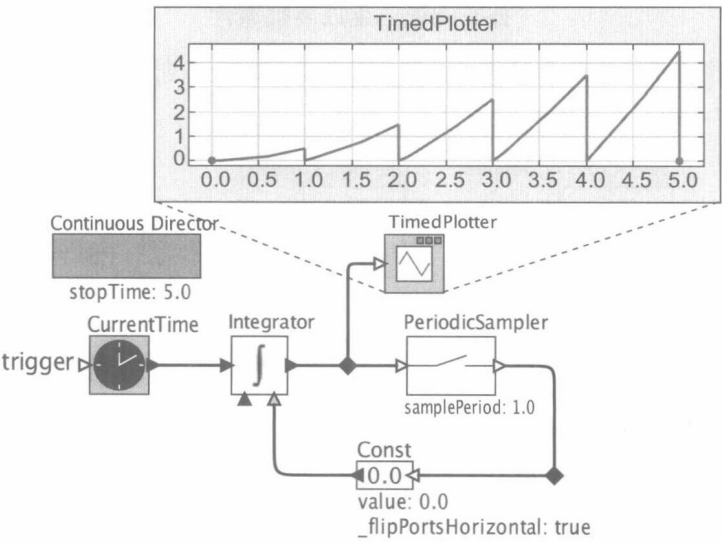


图 9-17 PeriodicSampler 角色，放在反馈回路中。在这种情况下，每隔一个时间单位 Integrator 将重置为 0，无论它的状态是什么

9.2.5 与 DE 互操作

连续域和 DE 域都支持离散事件信号。然而，在这些域之间有一个微妙但重要的区别。在连续域中，在求解器选择的时间戳，所有角色都点火。在 DE 域，只有当角色输入端口有一个事件或它以前要求点火时，该角色才能点火。因此，DE 模型可能更有效，尤其是当事件是稀疏的时。

结合这两个域来构建模型是有用的。这样的结合适用于许多 CPS 系统，例如，连续动态行为与基于软件的控制器相结合。构建连续域和 DE 域的混合模型比较容易，如下例所示。

例 9.7 考虑图 9-18 中的模型。模型的顶层在 DE 域中实现，并包括一个 Continuous (连续) 模型的不透明复合角色。这个示例对一个“作业车间”建模，这里作业的到达是离散事件，而处理速率由指数分布的随机变量给出，作业处理在连续时间中建模。

这个模型为每个作业分配一个整数。给出的整数通过给定的速率 (随机) 递增。速率越高，越快完成作业。当图中的虚线到达底图中的实线时，作业完成。上面的图显示了每个作业生成和完成的时间。注意，该模型有一个反馈回路，因此每次作业完成，一个新的作业从一个新的服务时间开始。

这个例子有点儿装，然而，从某种意义上来说，它通常不需要使用连续域 (见练习 4)。事实上，使连续时间信号线性增加或减少的模型通常可以单独在 DE 域中实现，而不需要求解器。但就是实际来说，构建混合域模型仍然有价值，因为它在连续的部分可以支持更复杂的动态。

如前面的例子所示，连续模型可以与放在 DE 模型内。相反，DE 模型也可以放在连续

例 9.8 考虑图 9-19 中的模型。该模型与例 7.14 中考虑的模型完全相同，除了它使用连续指示器而不是 DE 指示器。与 DE 指示器不同，连续指示器能够在给定的时间戳多次点火角色。因此，在它点火前，它不需要知道一个事件在复合角色输入时是否存在。这个指示器可以点火复合角色，从 DiscreteClock 获取一个事件，然后一旦事件已反馈，再次激活复合角色。

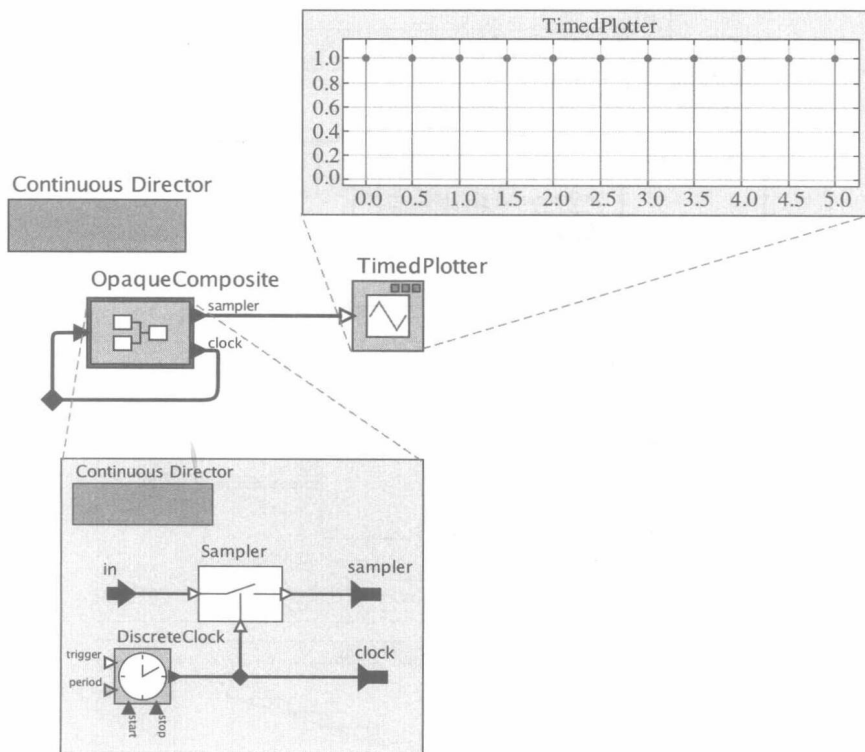


图 9-19 一个使用连续指示器可执行的离散事件模型，而不是使用 DE 指示器，如例 7.14 所示

9.3 混合系统和模态模型

混合系统 (hybrid system) 是连续动态行为与离散模式变化相结合的模式。这样的模型在 Ptolemy II 中使用模态模型 (ModalModel) 角色创建，ModalModel 角色在 Utilities 库中，并在第 8 章进行了解释。本节从讨论一个先构建的混合系统开始，并通过解释混合模型工作的原则结束。第 8 章解释了如何构造这样的模型，并说明了如何在模式细化 (mode refinement) 语句中处理时间。

例 9.9 弹力球模型如图 9-20 所示。它可以在 Tour of Ptolemy II (见图 2-3) 中 “Bouncing Ball” 下找到 (在 “Hybrid Systems” 项中)。弹力球模型使用一个命名为球模型 (Ball Model) 的模态模型组件。执行这个模型产生一个像这样的插图 (使用 GR (图形) 域构造的 3D 动画，本书不对 GR 域做介绍)。当球在空中时，这个模型是连续动态的；当球击中表面并弹回时，是离散事件。

图 9-21 显示了球模型的内容，它是一个有 3 个状态的模态模型：init、free 和 stop。在模态模型处于某个状态期间，其行为由模式细化指定。在这种情况下，只有 free 状态有细化，显示在图 9-21 的底部。init 状态是初始状态，只在状态传出转换的时刻使用，并用赋值

动作来初始化球模型。具体来说，转移标记如下：

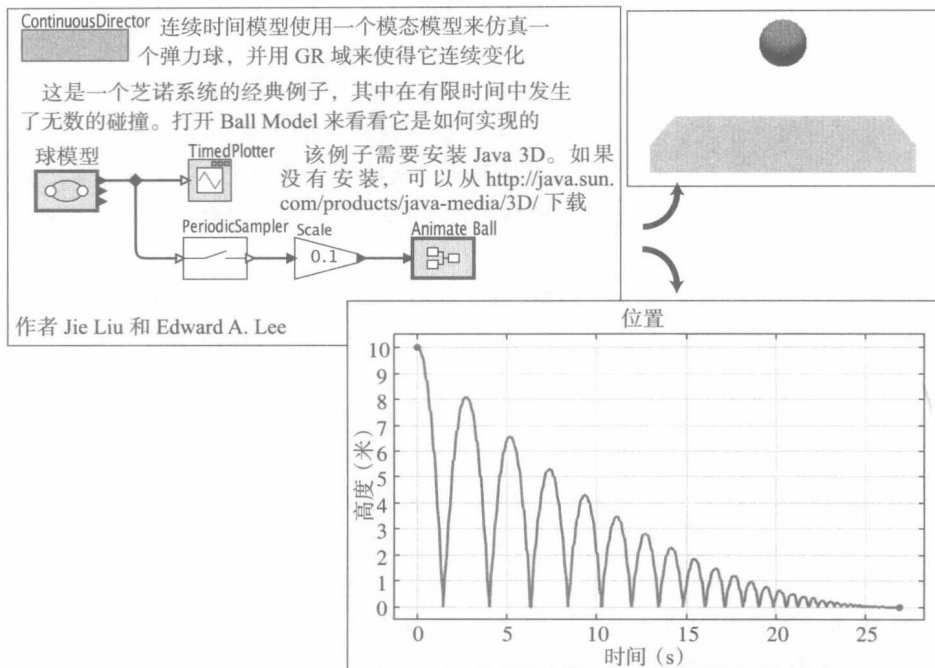


图 9-20 一个混合系统的例子：弹力球反弹的最高点

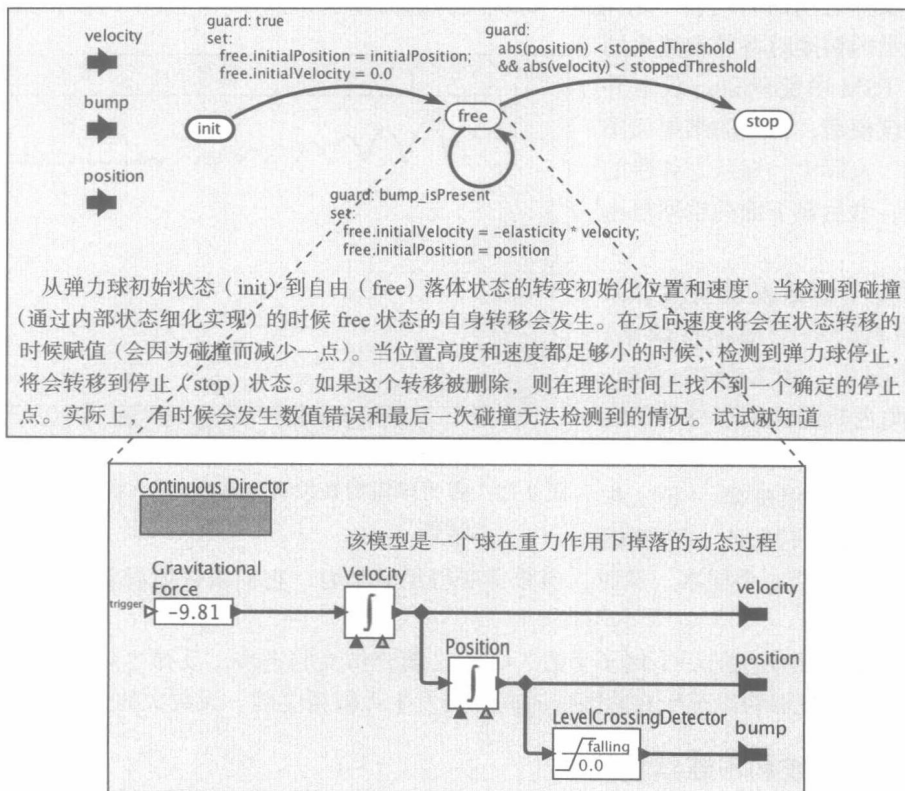


图 9-21 图 9-20 中球模型的内部

```

guard: true
set:
  free.initialPosition = initialPosition;
  free.initialVelocity = 0.0

```

第一行是一个条件，这是一个决定何时可以进行转换的判断。在这种情况下，总是进行转移，因为这个判断的值为 `true`。因此，该模型立即转移到模式 `free`（空闲）。这个转移发生在执行开始时的微步 0。“`set:`”行表明后面的行是赋值动作的定义（见 6.2 节）。第三和第四行设置目标模式 `free` 的参数。当球自由落体时，`free` 状态代表操作的模式，`stop` 状态表示球停止跳跃的模式。

当模型开始执行时，它处于 `init` 状态。由于 `init` 状态没有细化，当模态模型处于该状态时，球模型将没有输出。输出转移的条件总是有效的判定式，所以球模型将只在一个微步处于那个状态。

在 `free` 状态，细化代表万有引力定律，即物体有一个大概为 9.81m/s^2 的加速度。对加速度求积分得到速度，再次，求积分找到高度。在细化中，`LevelCrossingDetector` 角色用于检测什么时候球的高度为零。其输出在（离散）输出端口 `bump` 产生事件。图 9-21 表明，该事件触发一个状态转换回到同一个 `free` 状态，但是现在改变 `initialVelocity` 参数来颠倒符号，并通过 `elasticity` 参数来衰减速度的值。球反弹时它失去能量，如图 9-20 所示。

图 9-21 显示，当球的位置和速度小于指定的阈值时，状态机转移到没有细化的 `stop` 状态。这时，该模型没有产生输出。

弹力球模型说明了混合系统建模的一个有趣属性。事实证明，`stop` 状态是至关重要的。没有它，反弹之间的时间会不断减少，每次反弹的高度也是这样。在某个时候，当这些数字小于可表示的精度时将导致更大的错误。从 FSM 中删除 `stop` 状态并重新运行该模型，得到的结果如图 9-22 所示。实际上，球从它弹跳的表面降落，然后在下面的空间自由落体。

这里发生的错误说明混合系统建模中具有一个根本性缺陷。在这种情况下，由 `LevelCrossingDetector` 角色检测到的事件可能会被仿真器忽略。当这种事件发生时，这个角色和求解器一起工作试图确定精确的时间点。它确保仿真器在那时包括一个样本。然而，当数字变得足够小时，它们由数值误差控制，事件就被漏掉了。

弹力球是芝诺模型的一个例子（见 7.4 节）。随着仿真的进步，反弹之间的时间变小，而且它变小的速度足够快以至于在有限的时间内会发生无限精度的、无数次的无限反弹事件。

9.3.1 混合系统和不连续信号

从例 9.3 看出，输出是输入由不连续函数的角色可以创建不分段的连续信号。这可能导

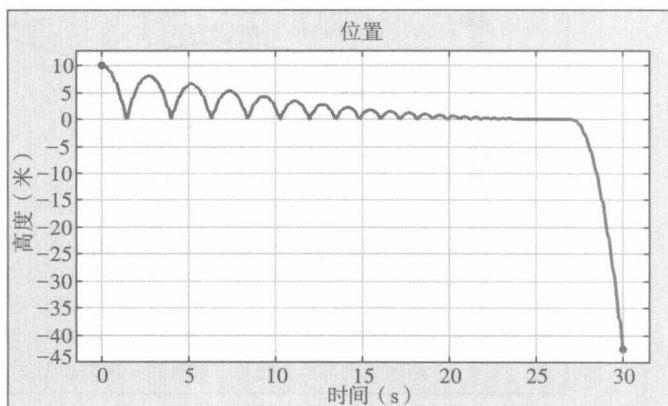


图 9-22 弹力球模型在没有 `stop`（停止）状态的情况下运行的结果

致求解器产生依赖行为，由求解器决定的任意步长强烈影响模型的执行。不过可以使用模态模型解决这些问题，如下例所示。

例 9.10 图 9-23 显示了图 9-10 中正确产生分段连续信号模型的一种变体。这个变体使用模态模型，它指定了在信号不连续时的转移。模态模型的转移是瞬时的，因为模型时间不改变。仅微步改变。在这个模型中，转移发生在时间 1.0 后的 `errorTolerance`（一个指示器参数）内。转移时，`zero` 状态的细化首先激活，在微步 0 产生输出 0，然后 `increment` 状态的细化在微步 1 激活，产生输出 2.0（或在 2.0 的 `errorTolerance` 内）。因此，输出信号是分段连续的。

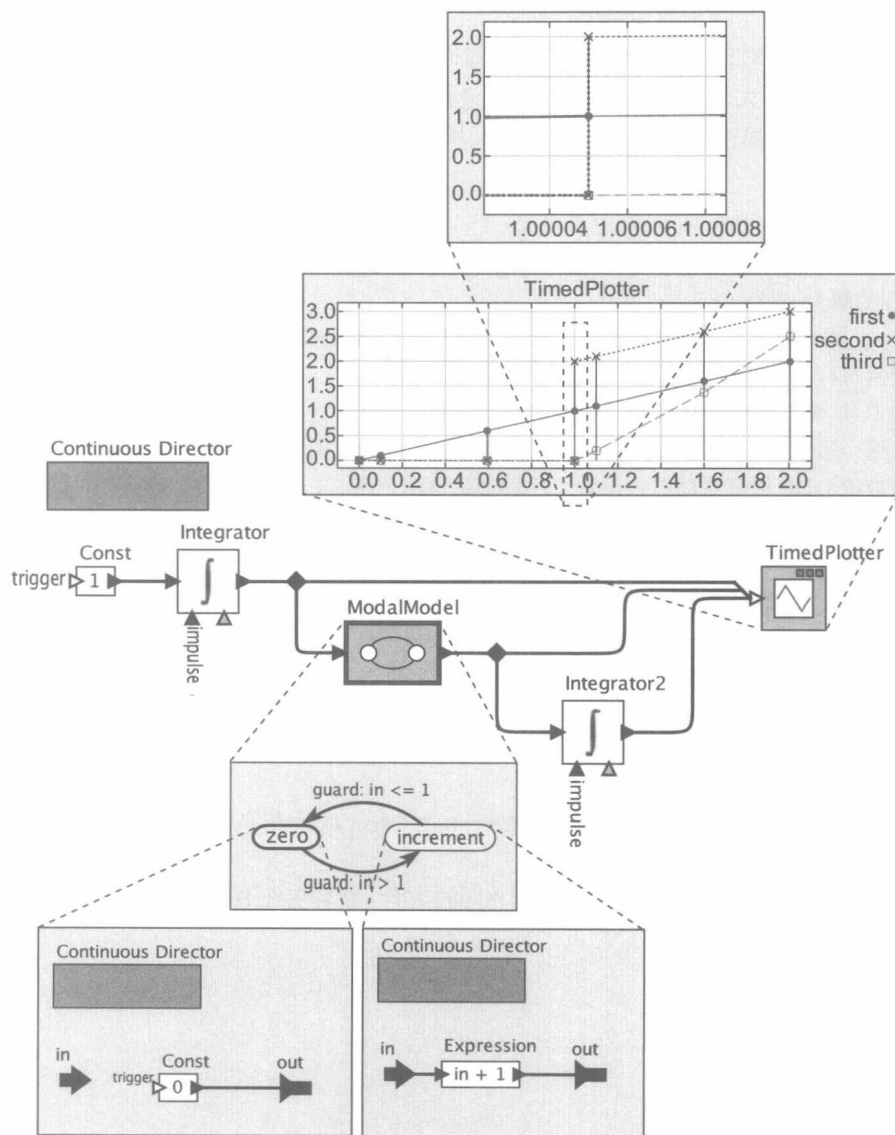


图 9-23 图 9-10 中模型的一种变体，它正确地产生了分段连续信号

模态模型角色的操作在第 8 章中说明。当与连续指示器结合时，这样的操作将自然而然地转换为混合系统的一个有效和有用的语义。为了完全理解模态模型和连续域的互操作，回

顾 6.2 节中描述的模态模型的执行语义是有用的。具体来说，模态模型的点火包括当前状态细化的点火（如果有的话）、计算条件的值，如果条件值为 `true`，则执行转移。同样重要的是，当模式处于不活跃状态下时，精化中时间将不前进。因此，在细化中的局部时间将滞后于在它环境中的全向时间。

在模态模型中，允许转移有输出动作（见 6.2 节）。这样的动作应该谨慎使用，因为转移可能发生在微步 0，由此产生的输出将不是分段连续的。如果输出动作用来产生离散事件，那么必须由来自分段连续信号的离散事件触发该转移。

9.4 小结

在数字计算机中对连续时间系统建模并逼近它们的物理行为是非常棘手的。Ptolemy II 的超密时间模型使得对大规模复杂系统准确地建模更加容易实现，对于混合连续和离散行为的系统尤其有效。本章描述的连续域，展现了利用这个时间模型提供复杂的建模和仿真的能力。

练习

- 令 x 是连续时间信号，其中 $x(0) = 1$ ， $\ddot{x}(t) = -x(t)$ 的， \ddot{x} 是 x 关于时间 t 的二阶导数。容易得出这个方程的解为 $x(t) = \cos(t)$ 。
 - 用 Integrator 角色来构建该信号 x ，而不使用其他角色或涉及三角函数的表达式。在合理的时间范围内绘制执行图来验证解决方案与理论预测是否匹配。
 - 修改求解器，指示器从使用 ExplicitRK23Solver 变为 ExplicitRK45Solver。定性描述结果的差异。哪个求解器提供更好的解决方案？“更好的”标准是什么？解释这个差异。
 - 所有数值 ODE 求解器都引入误差。尽管理论预测，解 $x(t) = \cos(t)$ 的幅值一直保持不变，但数值求解器不能实现。定性描述 ExplicitRK23Solver 和 ExplicitRK45Solver 如何执行，而保持指示器的其他参数为默认值。哪个求解器更好？标准是什么？
 - 用其他指示器参数做实验。errorTolerance 参数如何影响解？maxStepSize 呢？
- 例 9.2 说明使用 ContinuousTransferFunction 指定连续时间系统的传输函数。请证明在这个例子中给出的参数，对于图 9-4 和图 9-6 中的模型是等价的。提示：如果学习过一门典型的电气工程信号与系统的课程，那么这个问题很简单，但它是可行的，无需承认以下事实：如果信号 w 有拉普拉斯变换 W ，那么这个信号的积分对于所有复数 s 有拉普拉斯变换 W' ， $W'(s) = W(s)/s$ 。也就是说，在拉普拉斯域中除以 s 相当于在时域中积分。
- 考虑例 9.1 中的 Lorenz 吸引子。使用 DifferentialSystem 高阶角色实现相同的系统。给出你的 DifferentialSystem 的参数名和值。
- 例 9.7 中的模型实际上不需要连续域来实现相同的功能。请构造纯 DE 模型的一个等效模型。

计时系统建模

Janette Cardoso、Patricia Derler、John C. Eidson、Edward A. Lee、
Slobodan Matic、Yang Zhao 和 Jia Zou

本章致力于在复杂系统中为计时行为建模。将首先从系统时钟开始讨论，尤其强调多样时间。并将通过在 3 个特定的建模问题来说明如何使用多样时间。首先，考虑时钟同步，使用网络协议校正分布式系统中的时钟，以确保时钟以大致相同的速率运转。其次，考虑在系统行为中估算通信延迟的问题。最后，考虑在系统行为对执行时间所产生的影响估算问题。然后得出结论：介绍一种称为 Ptides 的编程模型，该模型可以让某些系统行为在计算和联网期间不受时间波动的影响，直到出现故障点。Ptides 计算模型能够使得信息物理融合系统 (cyber-physical system) 更具有确定性。

作为本章的前言，读者需要注意：在信息物理融合系统中讨论的时间模型可能非常混乱，因为在这样的模型中，时间本质上是多样 (multiform) 的。多种不同的观点和时间测量方式可以同时并存，比如“当”和“同时”这样的短语可认为是一个意思。

时间多样性的最明显来源是实际时间 (real time) 和模型时间 (model time) 之间的区别。“实际时间”的意思是模型在实际执行过程中所花费的时间，或被建模系统执行系统所需要的时间。如果模型的执行是对某个物理系统的仿真，那么“实际时间”可能指的是在仿真执行时流逝的外界统一概念的时间 (例如，你的笔记本电脑上运行程序时手表测量的时间)。相比之下，模型时间用在仿真中，并且其推进的速率与“实际时间”没有关系。

但是令人困惑的是，由于被仿真的物理系统可能是一个实时系统，在这种情况下，模型时间是实际时间的仿真，但与手表测量的实际时间不同。更糟糕的是，在信息物理融合系统的仿真中，可能有多种时间测量装置，而不是单一的手表。在分布式系统中，通过网络连接的多个微控制器有多个时钟。它们可能同步也可能没有同步，但即使它们是同步的，这个同步也会不可避免地存在缺陷，对缺陷的建模可能是模型的一个重要组成部分。因此得出一个结论就是，一个单一的模型可具有若干独立的时间轴来对应系统中的多个组件的运行。此外，如在第 8 章所讨论的，模态模型导致一些时间轴冻结，而让另一些运行。保持多个时间轴正常运行是一个大的挑战，这也将是本章讨论的重点。

10.1 时钟

如 1.7 节解释的，Ptolemy II 提供了一个跨域的一致性时间协同概念。Ptolemy II 支持多样时间。每个指示器都包含一个跟踪本地时间的本地时钟 (local clock)。本地时间通过指示器的 startTime 参数来初始化，并演变为一个给定的 clockRate。clockRate 是变化的。

一个简单的指示器的参数对话框如图 10-1 所示 (大多数指示器都不止这些参数，但每个指示器至少有这几个参数)。如果给定 startTime 参数，当模型初始化时，它指定本地时钟的时间。如果没有给定 startTime 参数，那么在初始化时时间将被设置为环境 (上层指示器，或层次

结构中上层的指示器) 时间, 如果这个指示器在模型的顶层, 其将设置为零。当指示器的本地时间达到 stopTime 所描述的值时, 指示器将请求不再点火 (通过 postfire 函数返回 false 值)。

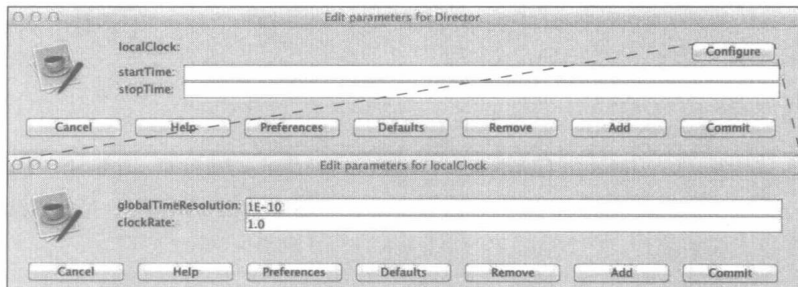


图 10-1 本地时钟的监视器参数

指示器的参数对话框中还包含一个配置本地时钟的设置 (Configure) 按钮, 如图 10-1 所示。这可用来设置时间分辨率 (time resolution) (见 1.7.3 节) 和时钟频率 (clock rate)。clockRate 参数主要描述本地时钟相对于上层指示器的时钟以何种速度前进。一个指示器把上层指示器的时间称作环境时间 (environment time)。如果没有上层指示器, 那么时钟的前进完全被这个指示器控制。

例如, 如果一个离散事件 (DE) 指示器有一个上层指示器, 则 clockRate 将从环境中得到的输入事件的时间戳转换为本地时间戳。如果它没有上层指示器, 那么所有事件都在本地生成, 而且指示器总是将时间提前到未处理事件的最小时间戳。

Ptolemy 中的每个指示器都有一个本地时钟。如果一个不计时的指示器 (如 SDF、SR) 没有上层指示器, 那么时钟值永远不会变化 (除非其 period 参数设置为非零值)。

例 10.1 图 10-2 展示了从时钟 c1 到时钟 c4 的 4 个不同时间轴。时钟 c1 (实线) 代表

一个与环境时间一致变化的时钟。时钟 c2 到时钟 c4 有不同的时钟频率、时钟值和偏移量。在前 5 个时间单位中, 所有的时钟以与环境时间相同的频率变化。c3 时钟从偏移量 -5.0 开始, 即迟于环境时间 5 个单位。当环境时间为 5 时, 则 c2 和 c3 的时钟频率发生变化: c2 的时钟频率增加, 而 c3 的时钟频率降低。c4 时钟被挂起, 所以它的值在接下来的 3 个时间单位内不发生改变。当时间为 8 时, c4 恢复。当时间为 10 时, 为了匹配环境时间, c3 的值被设置为 10。因为 c3 的时钟频率仍低于 1.0, 所以这个时钟立即开始滞后。

在 PtolemyII 中可以对这些不

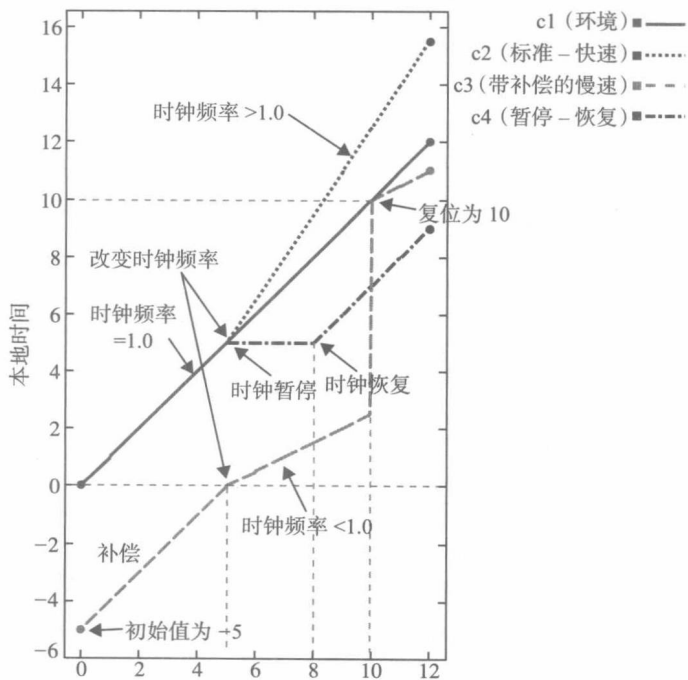


图 10-2 相对于环境时间, 时钟以不同的频率前进

同的时钟行为进行建模。可以针对时钟执行以下操作：定义一个偏移量（offset）、改变时钟值（clock value）、挂起（suspend）和恢复（resume）时钟、改变时钟频率（clock rate）。

例 10.2 产生图 10-2 中的曲线图的模型如图 10-3 所示。通过修改指示器的 localClock 参数中的 clockRate 参数来改变时钟频率。在右上角的 Fast（快速）复合角色中，设置该参数等于端口参数 rate，以便每一次新的频率提供给该输入端口，本地时钟频率都随之改变。RegularToFast 是一个离散时钟（DiscreteClock）角色，它在时间 0.0 时以时钟频率 1.0 开始，然后在时间 5.0 将频率改为 1.5。

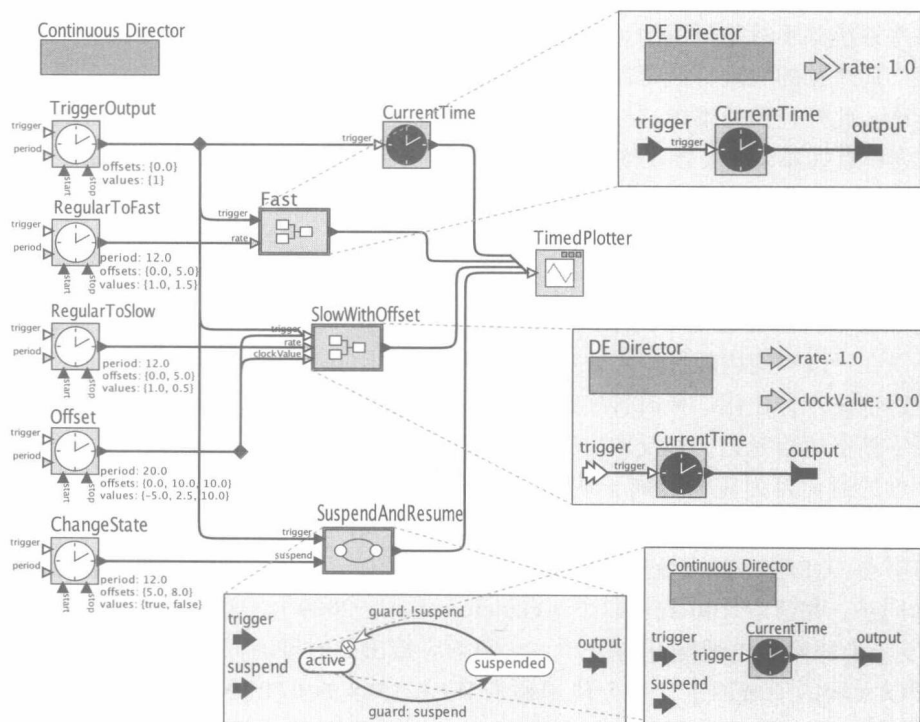


图 10-3 生成图 10-2 曲线图的模型

通过改变指示器的 startTime 参数值来更改时钟值。在仿真过程中，任何时间修改参数 startTime 时将会把时钟的当前值更改为 startTime 参数的值。在中间右边的 SlowWithOffset 复合角色有一个端口参数，称为 clockValue，其指示器的 startTime 设置为参照端口参数的表达式 clockValue。每当一个新值到达该端口参数时，就设置时钟值。左边的 Offset 角色在环境时间为 0.0 时将时钟设置为 -5.0，在环境时间为 10.0 时设置为 2.5，环境时间为 10.0 时设置为 10.0。后面的更新利用 Ptolemy II 中的超密时间模型立刻把时钟值从一个值改变为另一个值，用图 10-2 中的垂直虚线表示。

右下角的 SuspendAndResume 角色是一个模态模型（modal model），当状态机在不活跃状态时，连续指示器中的时钟被挂起，由图 10-2 中水平点画线（dash-dot line）线段表示。注意，进入活跃状态的转移是一个历史转移，目的是防止本地时钟重置为开始时间（如果给定开始时间）或环境时间（如果没有给定开始时间）。

指示器的本地时间可以通过使用将 useLocalTime 参数设置为 true（缺省值）的

currentTime 角色来画出。如果 useLocalTime 参数设置为 false，那么产生的输出将是模型顶层的环境时间。图 10-2 中的时间轴是通过定期触发这些 CurrentTime 角色获得的。

10.2 时钟同步

许多分布式系统依赖于一个统一的时间概念。提供一个统一的时间概念的蛮力技术是在通信网络中广播时钟。每当任何组件需要知道时间时，即可查看这个广播时钟。关于它的一个很好的实施范例是**全球卫星定位系统（GPS）**，其可以在 100 纳秒内进行多个分布式时钟的同步。该系统依靠部署卫星网络上的原子钟，通过仔细计算，并考虑到相对论效应。然而，GPS 对某些系统并不总是可用的（特别是室内系统），并且容易受到欺骗和人为干扰。更直接和自主的广播时钟的实现可能是昂贵并难以实施的，因为难以控制的通信延迟将导致生成的时钟不准确。此外，如何维护这样一个脆弱的系统也是一个很有意义的设计挑战，因为时钟源一旦成为单点故障，就会导致整个系统关闭（Kopetz, 1997; Kopetz and Bauer, 2003）。

一种更现代提高鲁棒性和精度的技术是使用**精确时钟同步协议（Precision Time Protocol, PTP）**来提供**时钟同步（clock synchronization）**。这样的协议通过交换带时间戳消息（每个时钟使用时间戳消息对自己频率进行小的修正）来保持松耦合的时钟网络同步。这种技术更加稳定，因为通信中的小故障并不产生太大影响，甚至当发生永久性通信故障时，时钟也可以在一段时间内保持同步，这取决于时钟技术的稳定性。

通常这种技术也比用广播时钟实现的同步更精确。对于大多数这样的协议，可以实现的精确度不依赖于通信延迟，相反取决于通信延迟的非称性（asymmetry）。也就是说，如果从 A 点到 B 点的通信延迟完全等同于从 B 点到 A 点的通信延迟，那么完全的时钟同步理论上是可能的。在实践中，这样的协议可以非常接近于在实际网络中的理论限制。例如，在欧洲核子研究中心（CERN）的白兔项目（The White Rabbit project），能够在跨越数公里的网络上实现时钟同步，精度在 100 皮秒以下（Gaderer et al., 2009）。这意味着，如果你同时访问两个由 10 千米的网络电缆隔开的时钟，它们响应给出的时钟时间相差不到 100 皮秒。在标准的基于以太网的局域网上，当今使用称为 IEEE 1588 的 PTP 协议在几十纳秒内实现精度是常见的（Eidson, 2006）。在开放的因特网上，通常使用一个称为 NTP（Mills, 2003）的 PTP 来实现几十毫秒的精度。

通常情况下，有一个或多个主时钟（并发生故障时重新选择），从时钟通过网络中的报文传递来同步到主时钟。保证了所有平台中时间的概念相同（即实现全局同步），具有良好定义的误差边界。下例模拟了不完全时钟同步的后果。

例 10.3 在电力系统中，传输线可能跨越数千米。当发生故障时，例如由于遭受雷击，所以查到故障位置的代价可能非常昂贵。因此，一种常见的方法就是：基于在传输线的每个观察点发现该故障的时间来评估。假定变电站 A 到变电站 B 之间的传输线长度为 60 千米。当发生故障时，两个变电站将经历一个可观察事件。假设电流以已知速率穿越传输线，那么变电站 A 观察到事件与变电站 B 观察到事件的时间差可以用来计算事件的位置。

令 X 表示沿着传输线的故障事件的位置（距变电站 A 的距离）。那么 X 满足以下方程，

$$\begin{aligned}s \times (T_A - T_0) &= X \\ s \times (T_B - T_0) &= D - X\end{aligned}$$

其中 T_0 是故障发生的时间（未知）， T_A 是在变电站 A 检测到故障的时间， T_B 是在变电站 B 检测到的故障时间， s 是沿着传输线传播的速度（光速）， D 是从 A 到 B 的距离， $D - X$ 是 B 到

故障位置的距離。將上述方程相減，得到 X 的解为

$$X = [(T_A - T_B) \times s + D] / 2$$

当然，如果两个变电站的时钟完全同步，那么这个计算是唯一正确的。假设变电站使用 PTP 来同步它们的时钟，变电站 A 是主设备。然后 A 和 B 将定期交换可以用来计算它们时钟差的报文。为了知道这是如何工作的，请参阅第 10 章补充阅读：精确时间协议或 Eidson (2006)。本书仅假定这种做法完美的 (A 和 B 之间的通信延迟是完全对称的)。只需关注在这个模型中使用这些信息调整对变电站 B 时钟控制策略的影响。图 10-4 中的模型显示变电站 A 定期将它的本地时间发送给变电站 B ，其中这个周期由 `syncPeriod` 参数给定，设置为 20.0 秒。

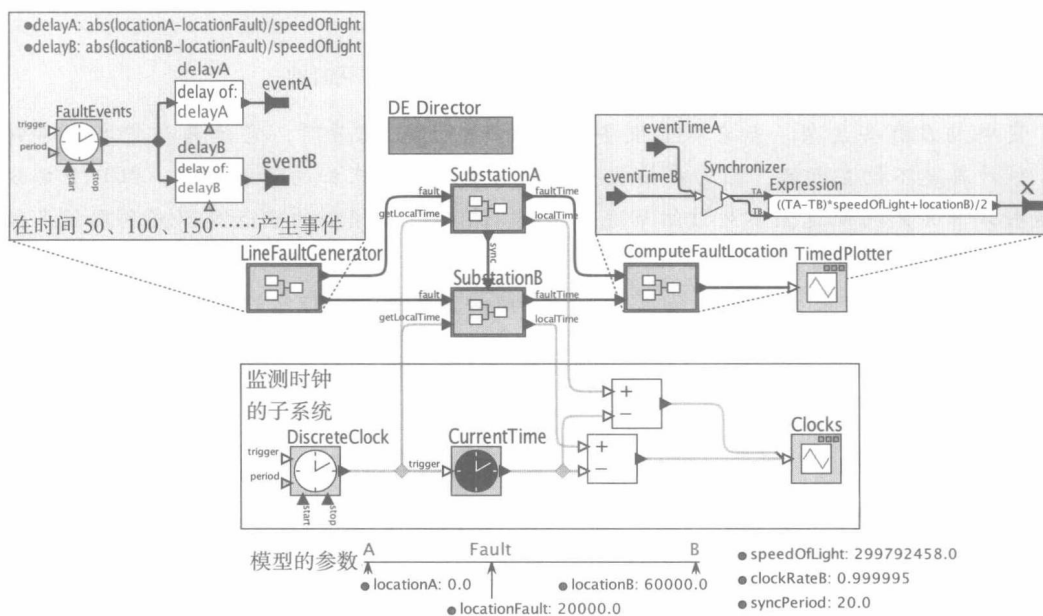


图 10-4 线路故障检测模型

在这个模型中，`LineFaultGenerator` 在 50、100、150 等时间产生故障，假设故障发生在离变电站 A 20 千米的位置。变电站角色将它们观察故障的本地时间发送给 `ComputeFaultLocation` 复合角色，它的任务就是使用上面的公式确定故障的位置。由于测量的故障时间在不同的时间到达 `ComputeFaultLocation` 角色，那么使用 `Synchronizer` (同步器) 等待，直到从每个变电站来的数据在计算之前已经收到。注意，如果一个变电站未能检测到这个事件并提供了一个输入，那么输入将产生错位，所以更现实的模型需要是更复杂的。

在图 10-5 和图 10-6 中描绘变电站模型。变电站 A 的模型非常简单，从上到下，对发送故障时间的 `fault` 输入进行响应，对具有本地时间的 `getLocalTime` 输入进行响应，并定期给 `sync` 输出发送本地时间。

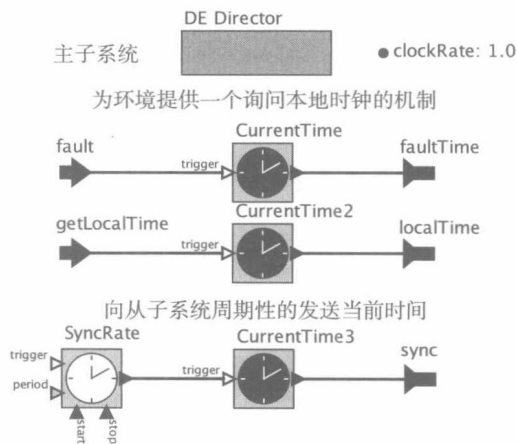


图 10-5 线路故障检测——变电站 A，主时钟

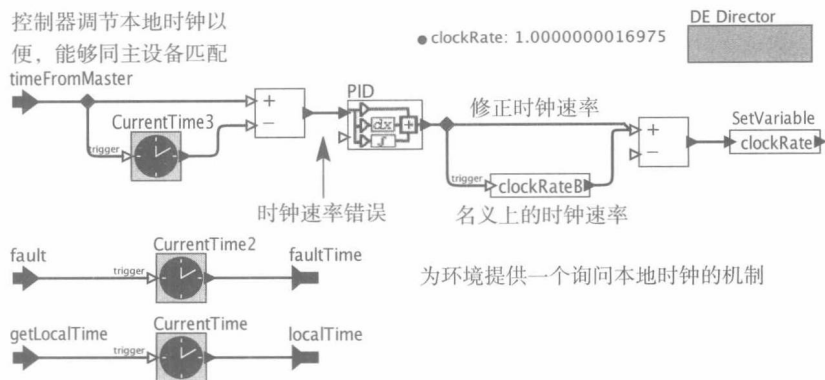


图 10-6 线路故障检测——变电站 B，从时钟

变电站 B 有些复杂。当它从主设备接收到一个 sync 信号时，它计算与它本地时钟的差。该计算是不切实际的，因为在接收 sync 信号时有一个未知的时间延迟，但 PTP 协议采用这种计算方式因此这里不对该细节进行建模。相反，该模型的重点是研究利用信息做什么，它使用 PID 控制器产生对本地时钟的校正。将这个校正添加到本地时钟频率，然后使用 SetVariable 角色存储在 clockRate 参数中。这个指示器时钟使用与它的时钟速率一样的参数，所以每一次校正，本地时钟的频率将改变一次。

图 10-7 显示了仿真结果。上图显示时钟中的误差。由于变电站 A 是主设备，它没有误差，所以它的误差是常数零。仿真开始时，B 的时钟相对于 A 线性偏离。在 20 秒时，B 接收第一个 sync 输入，PID 控制器提供了一个减少偏离速率的校正。在 40 秒时，另一个 sync 信号进一步减少偏离速率。下图显示在 50、100、150 等时对故障位置的估计。正确的故障位置是 20 千米，可以看到图中随着时钟的同步，估计位置收敛到 20 千米。

图 10-8 显示了如果没有时钟同步 (sync 信号从来没有到达 B) 将发生什么。在这种情况下，B 的时钟相对于 A 线性偏离，估计的故障位置的误差将无限增长。

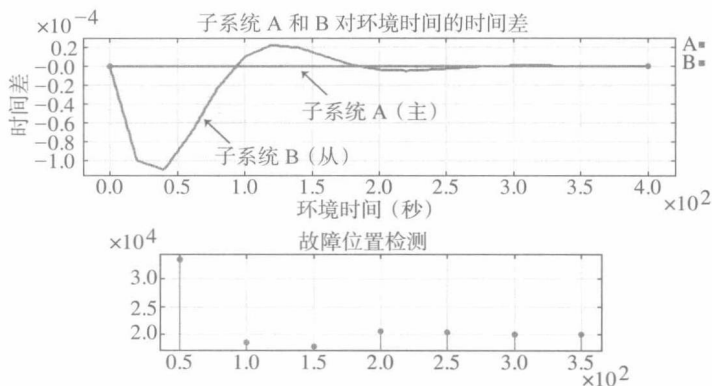


图 10-7 有时钟同步的线路故障检测

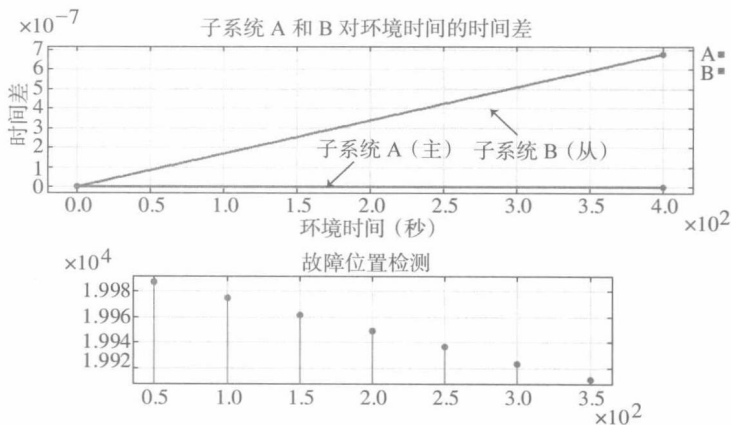


图 10-8 没有时钟同步的线路故障检测

10.3 通信延时建模

在设计-空间探索一个项目名: (design-space Exploration) 中, 设计师评估他们的设计是否能在给定的体系结构中良好运转。通信网络作为系统架构的一部分, 通信带来的延时对系统行为的影响, 因此在通信网络中引入了延时的概念。对彼此独立的常量通信延时建模是非常简单的, 但是考虑到共享资源情况就会很有趣 (和更实际), 会产生结果相关和可变的延时。本书先从简单模型开始, 然后发展到更加有趣的模型。

补充阅读: 精确时间同步协议

右图显示了一个典型的 PTP 是如何工作的。主时钟节点 A 发起报文消息交换。第一个包报文在 t_1 时刻发送 (由主时钟发出), 包含 t_1 的时间标记。从时钟 B 在时间 t_2 (由主时钟确定) 接收到该消息, 但由于从时钟无权访问主时钟, 所以从时钟根据自己的时钟得到接收消息的时间为 t_2' 。如果从时钟相对主时钟的偏差为 e , 那么

$$t_2' = t_2 + e$$

从时钟根据自己的时钟在 t_3' 时刻或根据主时钟的 t_3 时刻发送报文给主机作为响应, 所以

$$t_3' = t_3 + e$$

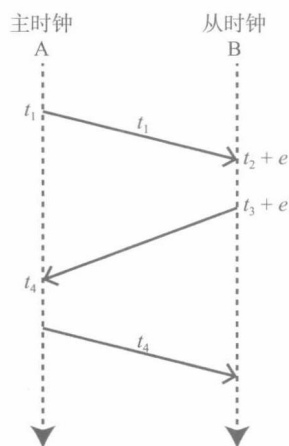
主时钟在时刻 t_4 接收到第二个报文, 并用通过发送包含 t_4 值的第三个报文进行应答。现在从节点有时间 t_1 、 t_2' 、 t_3' 和 t_4 。现在可得到往返通信传输延时 (在传输中报文从 A 到 B 和收到应答报文所花的时间) 是

$$r = (t_2 - t_1) + (t_4 - t_3) = (t_4 - t_1) - (t_3 - t_2)$$

在 B 节点, 尽管不知道 t_2 和 t_3 , 但上式也可以计算, 因为 $(t_3 - t_2) = (t_3' - t_2')$, 对 B 节点是已知的 t_2' 和 t_3' 的。如果通信延迟是对称的, 则单向延时为 $r/2$ 。为了修正节点 B 的时钟, 只需要估计 e 。这将告知 B 的时钟是否超前或延迟, 因此可以通过放慢或加快时钟来进行修正。如果通信信道有对称延迟 (即 $t_2 - t_1 = t_4 - t_3$), 就可以得到一个较好的估计,

$$\tilde{e} = t_2' - t_1 - r/2$$

事实上, 如果通信延迟是完全对称的, 那么 $\tilde{e} = e$, 是准确的时钟误差。节点 B 就可以通过 \tilde{e} 来调整它的本地时钟。



10.3.1 固定和独立的通信延时

在离散事件 (DE) 域中, 相互独立的网络延迟都可以使用 TimeDelay 角色来简单地建模。

例 10.4 图 10-4 中的线路故障检测器可理想化地计算变电站 B 的时钟误差。在实践中, 计算时钟差很重要。PTP 协议中实现的典型技术见第 10 章补充阅读: 精确时间同步协议, 并在图 10-9 的模型中实现。

在图 10-9 中, 变电站 A 定期启动用于计算时钟差的消息序列。首先, 在主时钟 t_1 时刻,

它将这个时间值发送给变电站 B。变电站 B 响应。变电站 A 用带有 t_4 时刻的报文来响应 t_4 时刻收到来自 B 的回复。当变电站 B 接收到最后这个消息时，它有足够的信息来估计其时钟与主时钟之间的差。(同步器 Synchronizer) 角色确保这个估计是收到所有必要的信息之后计算的。

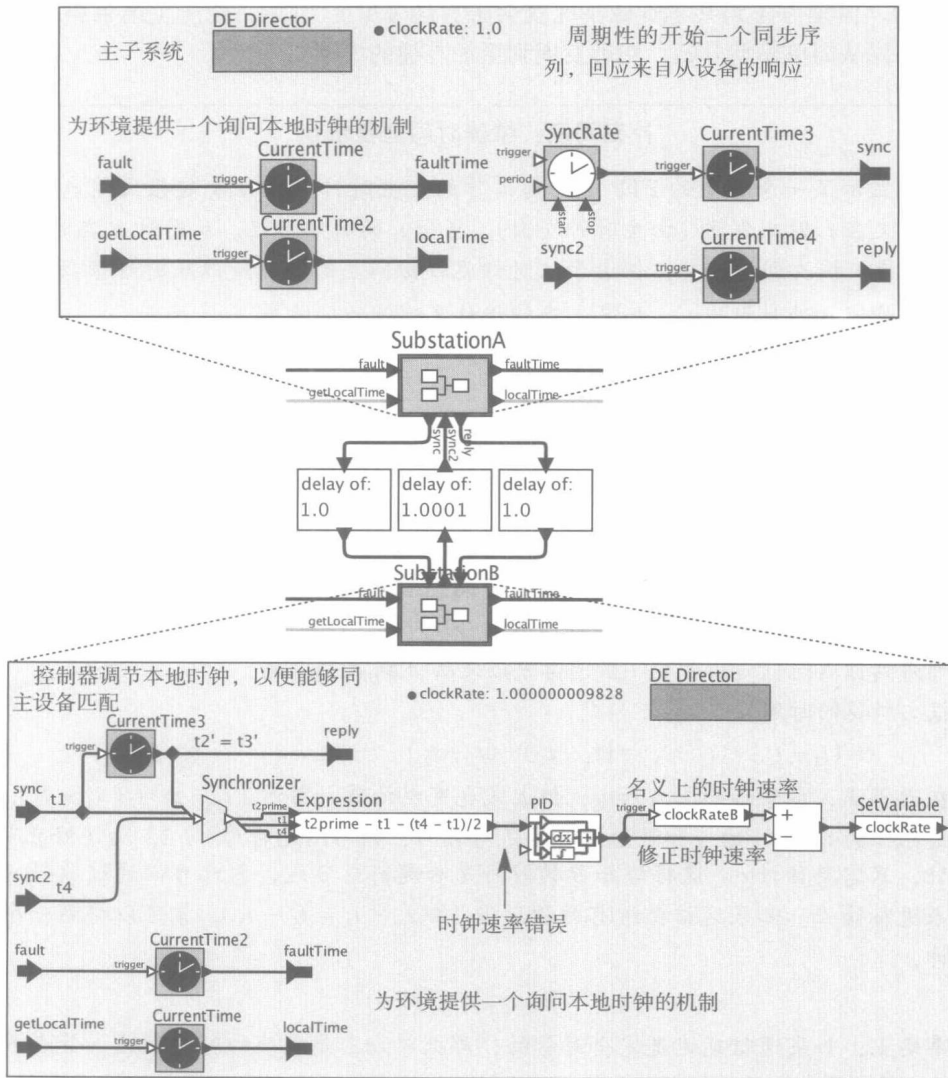


图 10-9 在 PTP 实现具有通信延迟的线路故障检测

图 10-9 有 3 个 TimeDelay 角色，这些角色可以为同步消息的通信中的网络延时建模。有趣的是，如果所有 3 个延时设置为相同的值，甚至是一个相当大的值，如 1.0 秒，那么确定故障位置的模型性能与理想化模型的性能基本上是相同的。但是，如果如图 10-9 所示结构，稍微改变一个延迟，为 1.0001，那么性能下降很大，如图 10-10 所示。从时钟误差进入稳态，且估计的故障位置收敛于大约 13 千米，完全不同于实际的故障位置 20 千米。显然，如果希望通信信道是不对称的，那么设计师必须努力改善控制算法。为 PID 控制器选择不同的参数可能有用，但有可能是以延长收敛时间为代价。

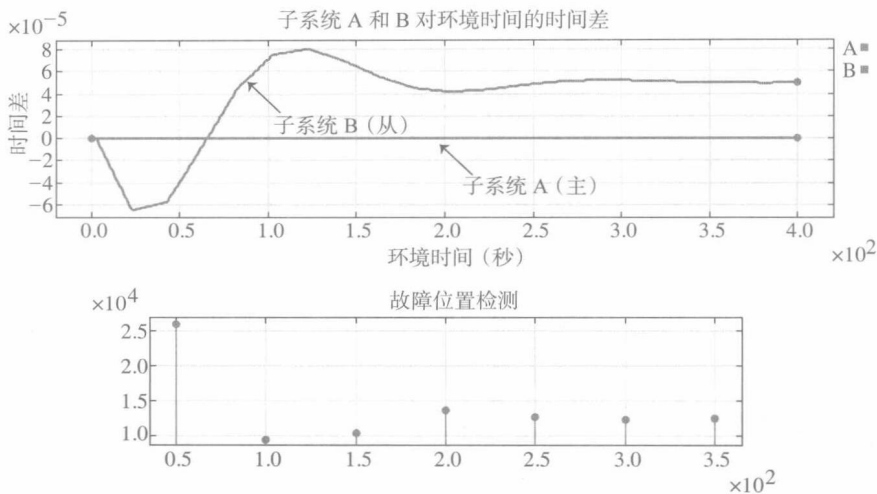


图 10-10 当网络延迟不对称时, 采用 PTP 执行线路故障检测性能不佳

10.3.2 共享资源竞争行为建模

在图 10-9 模型中, 角色之间的每个连接都有一个固定的通信延迟。对于实际的通信信道这是不现实的。其中延迟还与信道的其他用途有关。大多数信道资源(无线带宽、线路、路由器中的缓冲区空间等将是共享的), 并网络的延时会因为这些共享资源的其他用途而导致明显地变化。

可以修改图 10-9 中的模型对通过一个更详尽的网络模型对所有消息进行路由。然而, 这将导致建模复杂度的增加。例如, 假设我们希望用单一 Server (服务器) 角色为网络建模, 这也是最基本的共享资源的模型。然后, 通过信道的所有消息被合并成一个单独的数据流, 供给 Server 角色。在通过 Server 角色后, 这些流将再次分发, 所以目的地址应该在数据流进行合并前编码到信息中。因此, 该模型会变得相当复杂。

幸运的是, Ptolemy II 具有处理共享资源更简明的机制。面向切面建模 (Aspect-Oriented Modeling, AOM), 它基于面向切面编程 (Kiczales et al., 1997), 从功能映射到实现。这种结合了功能模型和实施模型还有调度的方法是由 Metropolis (Balarin et al., 2003) 提出的, 当时该机制称为数量管理 (quantity manager)。在 Ptolemy II 中, 一个切面 (aspect) 是一个管理资源的角色; 它与共享资源的角色和端口相关联。在仿真运行中, 这个切面角色安排资源的使用。资源和资源的用户之间的关联是通过参数实现的, 而不是通过端口的直接连接。因此, 切面被添加到现有的模型中而不改变现有模型的互连拓扑结构。接下来的例子显示了通信切面如何用来对共享通信资源行为进行建模。

例 10.5 图 10-11 显示的是一种线路检测器模型的变体, 其中已经被拖入模型的是称为总线 (bus) 的通信切面。在该例中, 变电站 A 与变电站 B 之间的 PTP 通信使用共享总线。在图中输入端口用注解 “Aspect: Bus” 对其进行标注, 并通过黑色对端口图标进行了填充。

总线有 serviceTime 参数, 该参数指定它获得令牌来遍历信道的的时间。在此期间, 总线很忙, 所以任何进一步尝试使用总线将被推迟。因此, 总线的作用就类似于具有无界缓冲区的服务器角色, 但由于它是一个切面, 所以不需要通过模型显式地表示所有穿过单个 Server

(服务器)实例的通信路径。

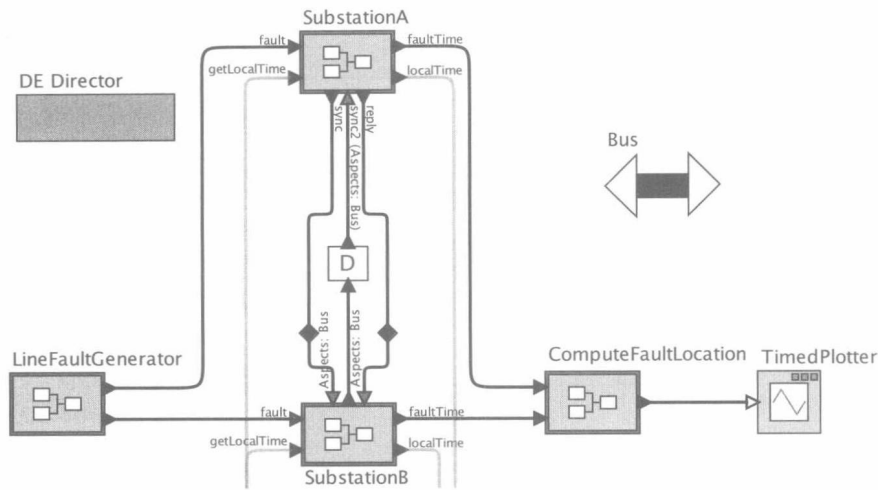


图 10-11 使用共享总线的通信线路故障检测，通过一个通信切面进行建模

在这个例子中，通信延时是对称的，因为没有对总线的竞争。因此，线路故障检测算法表现良好，行为类似于图 10-7。另一方面，如果在这个模型中允许其他通信路径使用此总线，比如在 ComputeFaultLocation 的输入端口，那么性能将大大降低，因为对总线的竞争将在通信延迟中引入不对称性。

使用一个切面进行通信建模，只需从库中拖入一个模型并给它指定一个有意义的名字。总线以及其他几个仍然在实验的（在撰写本书时），可以在 [MoreLibraries → Aspectslibrary] 库中找到。

一个切面是一种修饰符，这意味着它用参数给模型元素赋值（见第 10 章补充阅读：修饰符）。在总线情况下，它用 enable 和 messageLength 参数来修饰端口，如图 10-12 所示。当一个输入端口有可用的总线时，发送到该输入端口的消息将被延迟至少端口的 messageLength 参数和总线 serviceTimeMultiplicationFactor 参数的乘积的时间延迟总是存在的，因为如果当消息发送时总线忙碌，那么消息必须等到总线变为空闲。

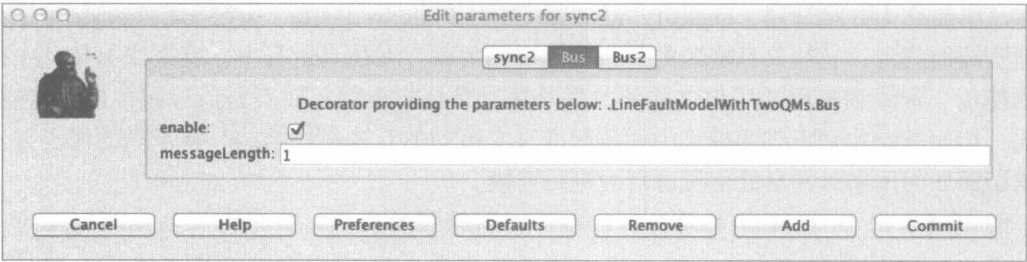


图 10-12 总线切面用 enable 和 messageLength 参数修饰端口。这个图显示了图 10-14 模型中的一个端口的参数编辑器，其中有两总线

可以添加任意数量的切面到一个模型中。每个都是一个修饰符，每个都可以独立启用。如果一个输入端口启用多个通信切面，那么这些切面根据它们启用的顺序调解通信。因此，方面是组合的。

补充阅读：修饰符

Ptolemy II 中的修饰符 (decorator) 是一个可以添加参数到模型中其他对象的对象，然后使用这些参数值提供一些服务。标准库中提供的最简单的修饰符是 **ConstraintMonitor**，可以在 **Utilities**→**Analysis** 库中找到。**ConstraintMonitor** 具有的属性，当它插入模型中时，将给模型中的角色中增加一个称为 **value** 的参数。**ConstraintMonitor** 跟踪模型中为所有角色设置的值的和，在其图标上显示这个和，并与 **threshold** 进行比较。

一个使用 **ConstraintMonitor** 的例子如图 10-13 所示，其中 **ConstraintMonitor** 被拖进一个带有 3 个角色的模型中，并更名为“**Cost**”一旦这个 **ConstraintMonitor** 在模型中，那么每个角色的参数编辑窗口将获得一个新的选项卡，如图的顶部所示，其中选项表上的标签与 **ConstraintMonitor** 的名称相匹配。用户可以为模型中的每个角色输入 **cost**，**ConstraintMonitor** 将在图标中显示总 **cost**。

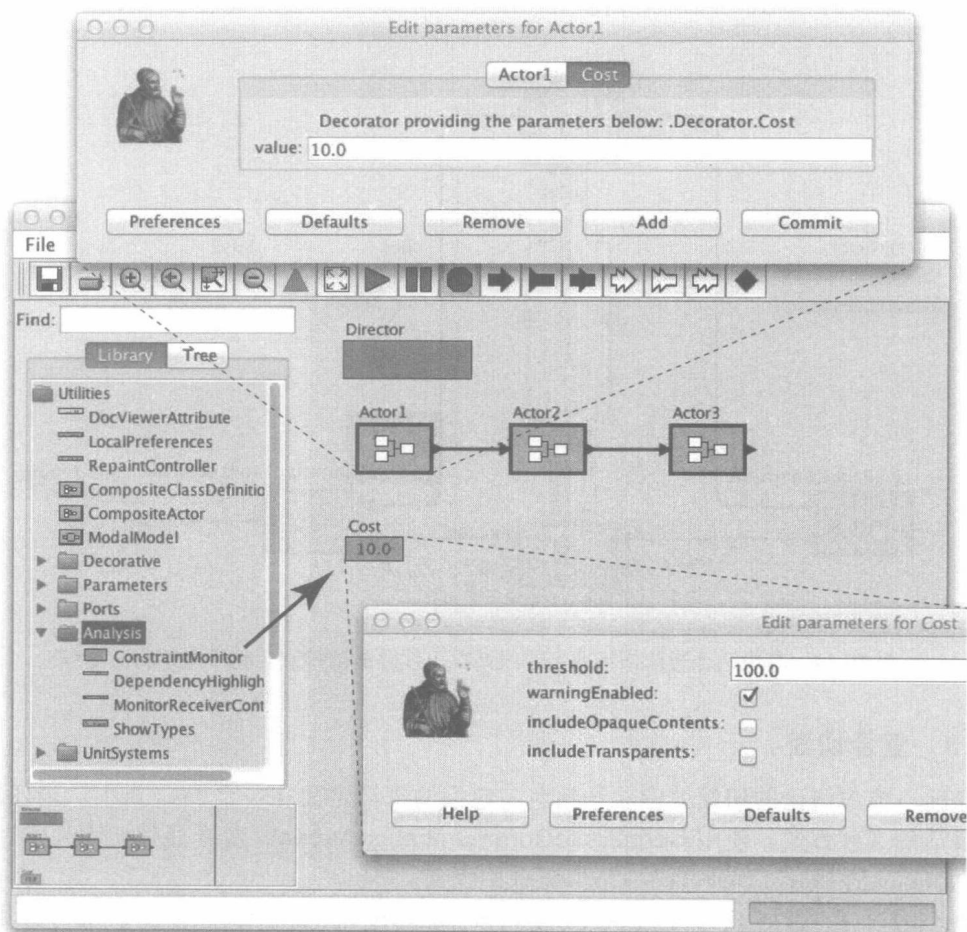


图 10-13 Ptolemy II 中的修饰符是一个可以添加参数到模型中其他对象中的对象，然后使用这些参数值提供一些服务。在这个例子中，**ConstraintMonitor**（已更名为 **Cost**）是模型中 **cost** 组件的总和，并与阈值 100.0 进行比较

ConstraintMonitor 有一个 threshold 参数，它是和的限制值。当和接近限制值时，ConstraintMonitor 图像显示将变为黄色。当和达到或超过限制值时，如果 warningEnabled 参数为真，那么给用户发出警告。threshold 的默认值是 Infinity，意味着没有限制。

ConstraintMonitor 还有两个其他参数，如图 10-13 底部所示。如果 includeOpaque Contents 参数为 true，那么不透明复合角色内的角色将被修饰。否则，不会被修饰。如果 includeTransparents 参数为 true，那么透明复合角色将被修饰。否则，不会被修饰。

修饰符还有许多其他用途。一个指示器可以作为一种修饰符。本章描述的切面是另一种修饰符的。

例 10.6 在图 10-14 中，第二个总线已经添加到模型中，从变电站 B 到变电站 A 的通信穿过 Bus 和 Bus2，依次，正如读者从到变电站 A 的 sync2 输入端口上的注释看到的一样。因此，通信延迟变为不对称的，线路故障检测算法表现不佳，产生的结果类似于图 10-10。Bus2 的 decoratorHighlightColor 参数由深灰色变为浅灰色，导致这端口和总线图标显示浅灰色。

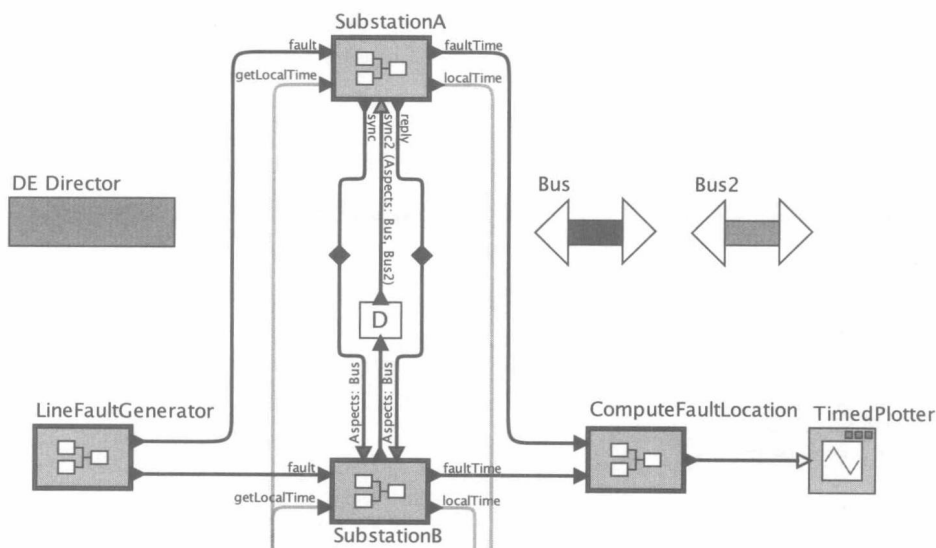


图 10-14 其中一个通信横穿两条总线情况下线路故障检测，将产生不对称通信

10.3.3 复合切面

在前一节中讨论的切面就像原子角色，它们的逻辑定义在一个 Java 类中。描述通信切面更灵活的一种方式，使用 CompositeCommunicationAspect (复合通信切面)。该角色可以在 Ptolemy 的 [MoreLibraries → Aspects] 中找到。

例 10.7 图 10-15 展示了使用 CompositeQuantityManager 实现总线的例子。在这种情况下，使用带有 Server (服务器) 角色的离散事件子系统对总线行为建模。输入总线的请求将在到服务器的输入队列中排队。当服务器变为空闲时，队列中的第一个输入有一个 serviceTime 的延迟。这个行为与原子总线切面的行为相同。

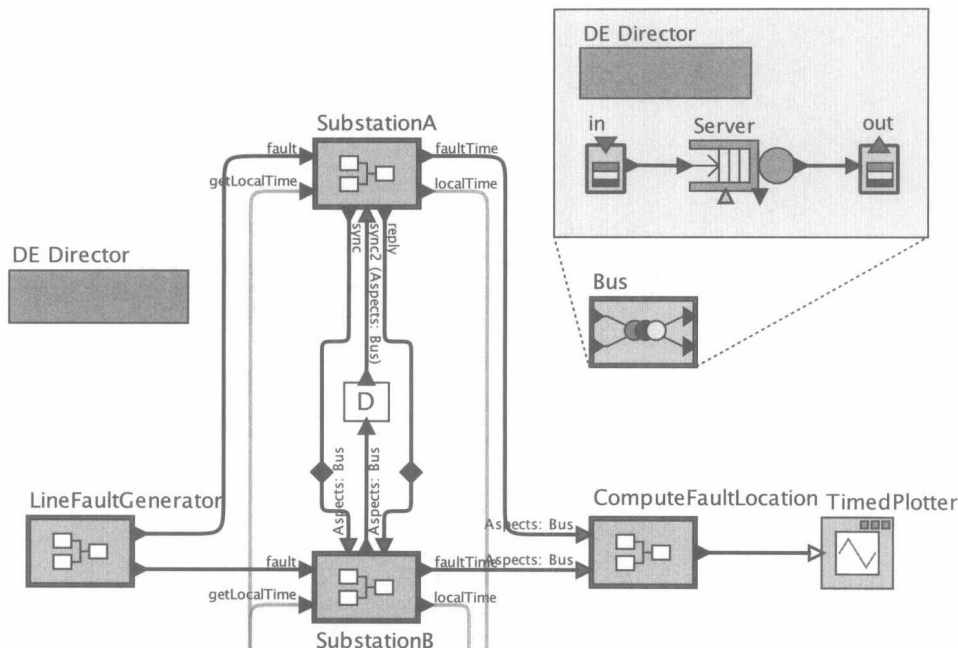


图 10-15 一个复合通信切面的总线

由于复合通信切面是一个简单的 Ptolemy II 模型，所以在设计上有很大的自由。

例 10.8 图 10-15 中的例子，使用的总线不仅仅实现变电站 *AB* 之间的通信，而且还用于与 *ComputeFaultLocation* 角色的通信。总线竞争使得通信延迟不对称，降低时钟同步的性能，并导致计算故障位置的性能变差。

如图 10-16 所示，在图中，对总线进行了修改，能够更好地提高网络性能。所以它现在是一个带有两个输入端口和两个不同服务器的更复杂网络。通过将通信路由到两个服务器，可以减少竞争。（每个输入端口涉及通信指定的输入端口，*in1* 或 *in2*，控制通信的切面。图的下方显示了如何使用端口修饰符参数来选择切面的端口）在图中，*ComputerFaultLocation* 的上方输入端口使用 *in2*。如果底部端口也使用 *in2*，处理变电站 *A* 与变电站 *B* 之间通信的端口使用 *in1*，那么竞争的减少足以提供更好的性能，类似于图 10-7。

10.4 执行时间建模

除了对网络特性（如通信延时）建模外，可能也需要对执行时间进行建模，执行时间是在一个特定执行平台上实现角色功能所需要的时间。在实现平台的模型上，对应用程序的功能和性能联合建模，*design-space exploration* 是一个非常强大的工具。这使得容易理解在网络基础设施和处理器架构中不同选择的作用。

在离散事件模型中，对于每个执行资源（如处理器），执行时间可以使用服务器角色来模拟，这里服务时间就是执行时间。例 7.5 和图 7-8 说明了一个简单存储系统的执行时间。然而，这样的模型很难与复杂功能的模型相结合。

幸运的是，Ptolemy 提供了执行切面，与通信切面一样，提供一种面向切面建模的形式。执行切面可用于对需要执行一个应用程序模型资源的竞争建模。这个机制类似于通信切面的机制，如下例所述。

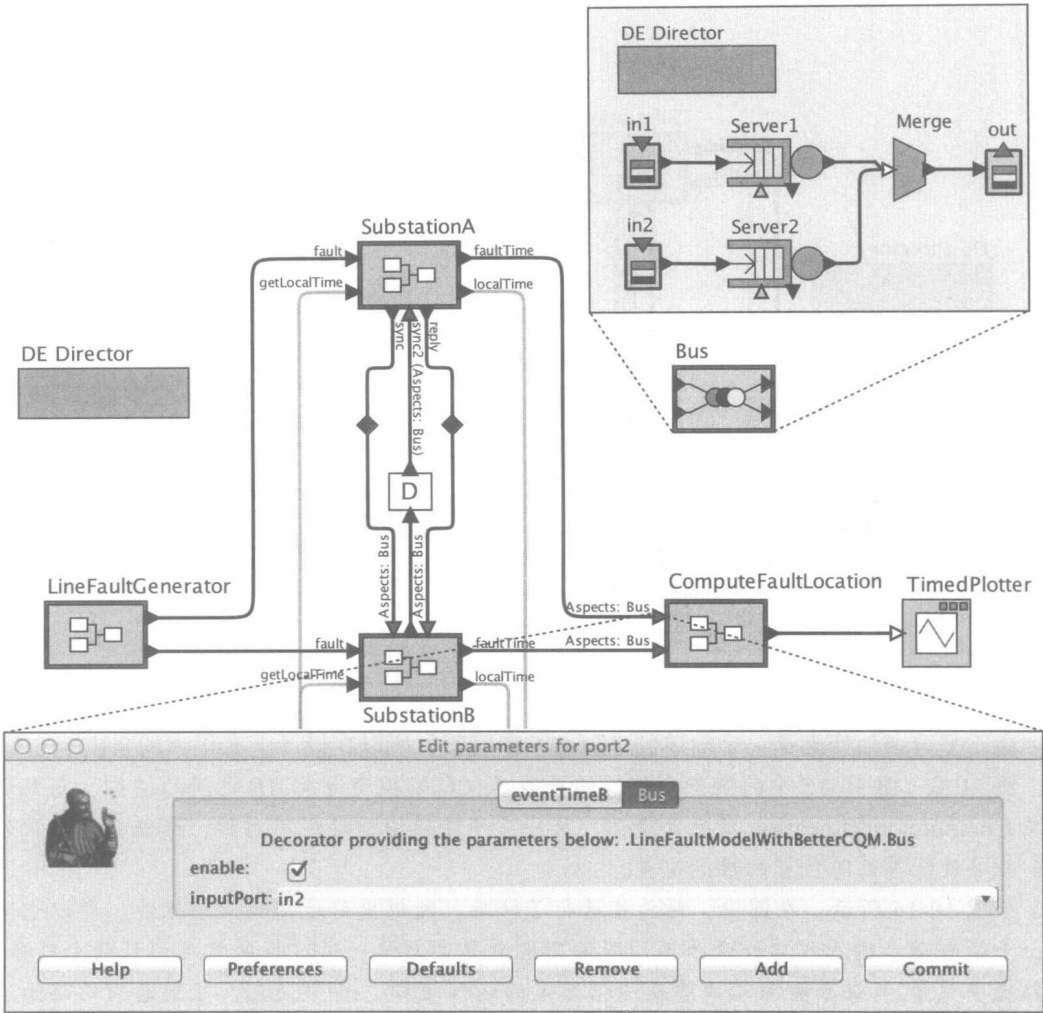


图 10-16 利用复合来减少资源争用的网络。显示了如何使用端口的修饰符参数选择处理通信的切面端口

例 10.9 1.9 节发电机模型的一个变体如图 10-17 所示。该模型包括两个可能的实现平台，一个平台有一个处理器，另一个平台有两个处理器，显示在图的底部。这两个实现平台使用 **CompositeResourceScheduler** 角色建模，在 **MoreLibraries** → **ResourceScheduler** 中。

在这个模型中，监督器（Supervisor）和控制器（Controller）角色执行两个处理器架构之一。选择哪一个由模型中的 **useTwoProcessors** 参数值决定。如果该参数的值为 **true**，那么将使用 **2Processor** 切面来执行监督 **Supervisor** 和 **Controller**。否则，将仅使用 **1Processor**。

当执行这个模型时，行为随着 **useTwoProcessors** 的值变化。当同时使用两个处理器时，没有资源竞争，因为监督器和控制器可以同时执行，与使用图中右下角的两个 **Server**（服务器）角色建模一样。然而，当只使用一个处理器时，监督器和控制器竞争同一个处理器，与使用左下角单一服务器建模一样。在这种情况下，两个反馈回路中的一个有更多的延迟，从而改变了模型的动态。特别是，当选择某些参数和试验条件时，处理器架构的选择可能影响图 1-11 中的过电压保护条件是否被触发。

注意在复合执行切面中使用 **RecordDiassembler** 角色。复合切面中子模型的输入是包含

请求执行资源的角色的修饰符参数 `executionTime` 值的记录。这个执行时间从该记录中提取，并成为该服务器的服务时间。

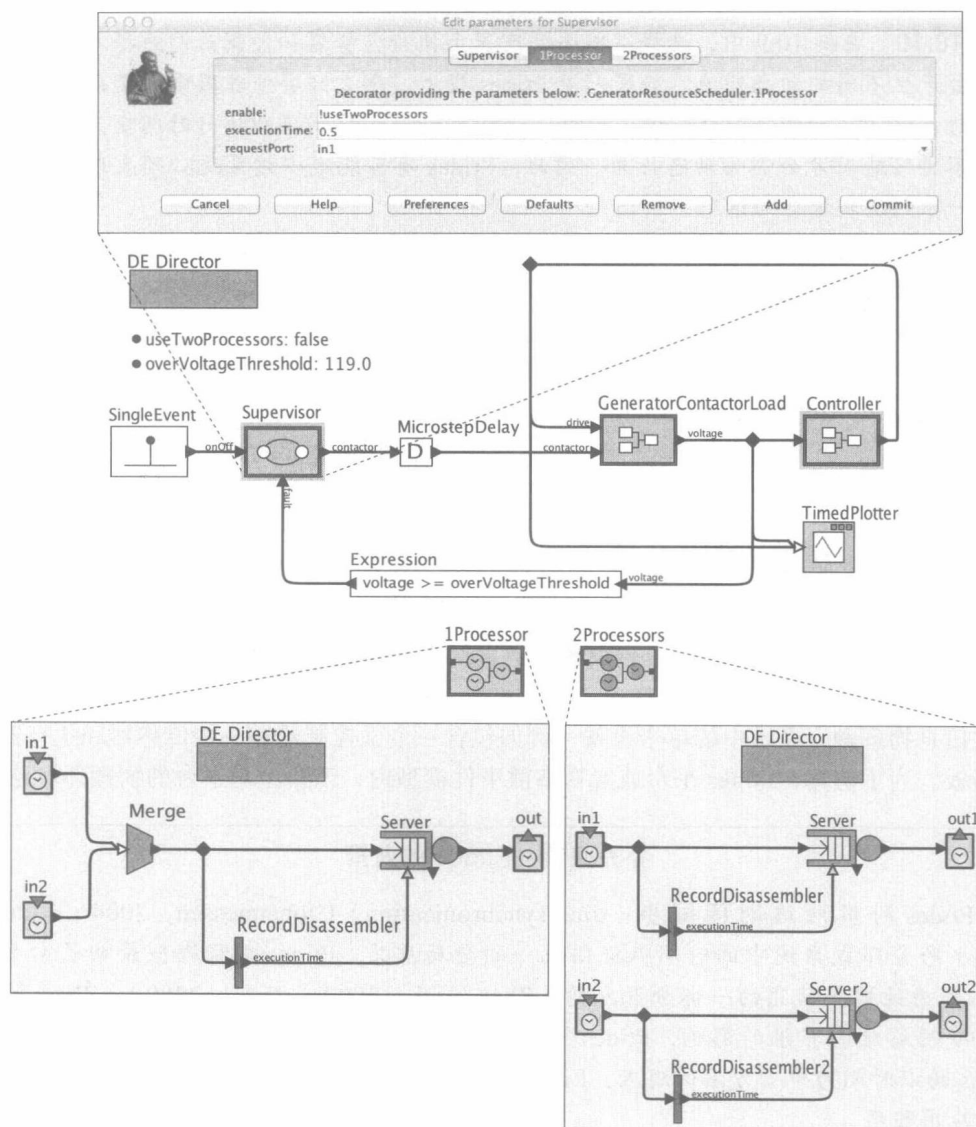


图 10-17 带有两个可供选择的执行切面的模块，一个为单处理器执行平台建模，另一个为双处理器执行平台建模

10.5 分布式实时系统的 Ptides 模型

目前为止，本章的重点是对系统实现中的计时行为进行建模和仿真。时间模型的另一个作用是 `specify` 计时行为。也就是说，计时模型可以给出一个完整需求而不需要完全描述实现。为此，本章后面将重点讨论分布式实时系统的编程模型 `Ptides`[⊖]。

[⊖] 这个名字来自“programming temporally integrated distributed embedded systems”的缩写。

Ptides 模型用来解决上面例 10.9 的问题，其中系统的行为取决于执行该系统的硬件和软件平台的细节。Ptides 的一个关键目标是为每个正确的系统执行都提供完全相同的动态行为。

例 10.10 在例 10.9 中，选择在单个处理器上执行监督器和控制器与选择在两个处理器上执行会产生不同的动态行为。如果采用 Ptides 模型，那么只要处理器资源足以提供一个正确的执行，这两个行为将是相同的。此外，直到监督器和控制器的执行时间变为足够大使得执行将不再可能时才会影响动态行为。因此，Ptides 有可能减少该系统必须表现细节的敏感度。在一系列实现中行为达到一致。

Ptides 模型是对时间戳具有一定制约的离散事件模型。Ptides 用于设计事件点火的分布式实时系统，其中事件可以定期发生（如数据采集系统）或不定期发生。与离散事件不同，Ptides 的关键思想是，时间戳与传感器和驱动器（这是桥接信息物理融合系统中网络与物理部分之间的设备）的实际时间有关联。Ptides 的第二个关键思想是，它利用网络时间同步（Johannessen, 2004；Eidson, 2006）给分布式系统中时间戳提供统一的全局意义。关于 Ptides 最有趣的、微妙的并有可能混淆的部分是多个时间轴之间的关系。不过，这种关系也有它的价值。

10.5.1 Ptides 模型的结构

Ptides 模型包括一个或多个 Ptides 平台（Ptides platform），其中每个平台都对网络上的一个计算机建模。Ptides 平台是**复合角色**，包含代表传感器、驱动器和网络端口的角色以及执行计算和更改时间戳的角色。Ptides 平台包含一个 Ptides Director（指示器名），并表示在分布式信息物理融合系统中的单个设备，例如包含一个微控制器和一些传感器和执行器设备的电路板。为了仿真，Ptides 平台放置在离散事件模型内，该模型对平台的物理环境建模。

补充阅读：Ptides 的背景

Ptides 利用**网络时间同步**（time synchronization）（Johannessen, 2004；Eidson, 2006）给分布式系统中的时间戳提供统一的全局意义。Ptides 编程模型最初是由 Yang Zhao 作为她博士研究的一部分开发的（Zhao et al., 2007；Zhao, 2009）。Zhao 表明，假设受网络延时界限的影响，Ptides 模型是确定性的。Lee et al.（2009b）阐述了类似 Ptides 的以时间为中心方法的观点，Eidson et al.（2012）给出了发电厂控制的 Ptides 概述和应用程序。

已实现的前期工作如下：在 Derler et al.（2008）提出了一个模拟器，Feng et al.（2008）和 Zou et al.（2009b）提出了一个适用于嵌入式软件系统实现的执行策略后。Zou（2011）开发了 PtidyOS，一个在嵌入式计算机上实现 Ptides 轻量级微核，和从模型产生嵌入式 C 程序的代码生成器。Matic et al.（2011）改写了 PtidyOS 和代码生成器以便说明它们拟应用在智能电网中的。

Feng and Lee（2008）用增量检查点扩展了 Ptides，以提供容错措施。他们发现还原可以从错误中进行卷回恢复，Ptides 中的关键约束是驱动器的动作不能被卷回恢复。Ptides 已经用来协调用 Java 编写的实时组件（Zou et al., 2009a）。

谷歌为管理分布式数据库独立开发了一个与 Ptides 类似的技术（Corbett et al.,

2012)。在这项工作中，时钟通过数据中心达到同步，数据中心之间发送的消息是时间戳。该技术提供了数据库访问和更新的确定性和一致性的测量。

假设满足网络延时界限，Ptides 的正确实现是确定性的，因为从传感器获得的时间戳事件的序列总是导致唯一的和良好定义的时间戳事件序列提交给驱动器。但是，这种确定性不保证该事件准时提交给驱动器（在时间戳的截止时间之前）。决定事件是否可以按时交付给驱动器的问题称为**可调度**（schedulability）问题。即给定一个 Ptides 模型，是否存在一个在截止时间前角色点火的调度表。Zhao (2009) 针对某些模型解决了该问题。该问题被 Zou et al. (2009b) 进一步讨论，并在很大程度上被 Matsikoudis et al. (2013) 解决。

例 10.11 如图 10-18 所示是一个简单的 Ptides 模型。该模型有两个平台，分别连接一个物理设施（通过传感器和驱动器）和一个网络。顶层指示器是一个 DE 指示器，而平台指示器是 Ptides 指示器。物理设施可能本质上是一个连续（Continuous）模型。

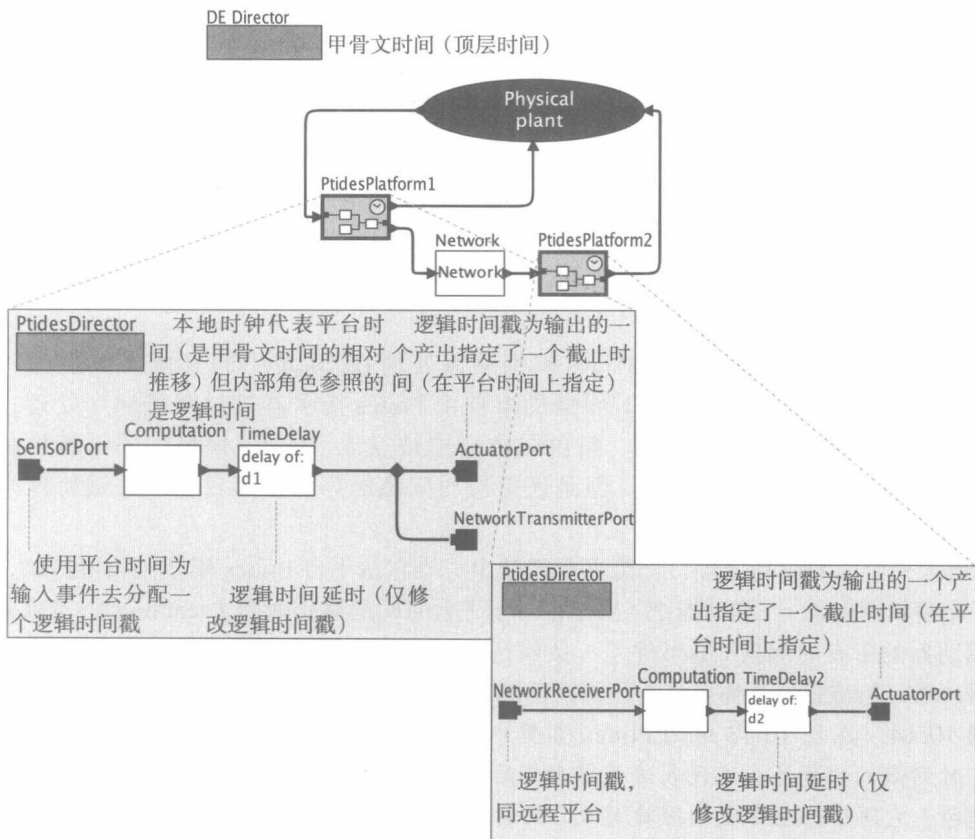


图 10-18 有两个 Ptides 平台、传感器、执行器和网络端口的 Ptides 模型

Ptides 模型利用了 Ptolemy 的**多样时间**（multiform time）机制。这种模型中的常见模式假设在模型层次结构顶层的时间轴表示整个系统均匀前进的理想物理时间轴。这个时间轴不能通过网络中的计算装置直接观察，相反必须用时钟近似地测量。将顶层中这样的理想时间作为**甲骨文时间**（oracle time）。在 MARTE 时间库中，物理时间的相同理想概念简称为**理**

想时间 (ideal time)(André et al., 2007)。

在平台中,本地时钟维持称为平台时间 (platform time)的时间轴,它接近甲骨文时间。平台时间是精密计时的 (chronometric),是甲骨文时间的一个不完美的测量。Ptides 模型的编译器可能认为平台时间完全跟踪 oracle 时间,或更有趣的是,对跟踪中的缺陷和网络中的差异建模,见 10.2 节的说明。

例 10.12 在图 10-18 中的 Ptides 模型中,顶层的指示器时钟表示甲骨文时间。Ptides 指示器的时钟表示平台时间。彼此平台间偏移量可以进行参数化,因此平台时间也会有偏移。

Ptides 的关键创新是,称为逻辑时间 (logical time)的第二条时间轴在平台中起着重要作用。分布式系统中逻辑时间的概念由 Lamport et al. (1978)提出,并应用于 Ptides 来实现分布式实时系统中的确定性。将给任何请求 Ptides 指示器当前时间的角色提供逻辑时间,而不是平台时间。唯一能访问平台时间的角色是传感器、驱动器和网络接口,即信息物理边界上的角色。具体来说,当传感器在 Ptides 模型中产生一个事件时,事件的时间戳是等于传感器测量的平台时间的逻辑时间值。也就是说, Ptides 将逻辑时间与传感器中的物理时间 (由平台时间测得的)绑定。

例 10.13 在图 10-18 中的 Ptides 模型中,传感器使用 PtidesPlatform1 的平台时间为它产生的每个事件构建一个时间戳。这样的—个事件表示物理设施的一个测量,它的 (逻辑)时间戳等于时间的本地测量。

令 t_s 为传感器测量的平台时间。由该传感器角色产生的事件将有逻辑时间戳 t_s 。

然而,TimeDelay 角色以逻辑时间运行。它对逻辑时间戳进行简单地操作。平台时间对它不可见。如果延迟值为 d_1 的 TimeDelay 角色的输入有 (逻辑)时间戳 t ,那么它的输出将有时间戳 $t + d_1$ 。

一旦传感器产生一个事件, Ptides 模型就像处理 DE 系统中的其他离散事件一样处理该事件。也就是说,具有逻辑时间戳的事件由 Ptides 指示器按时间戳顺序处理,而不特别涉及平台时间或甲骨文时间。角色在仿真中被点火。Ptides 模型的关键特性是,尽管有分布式架构和不完美的时钟,但是还是按时间戳顺序处理事件。该关键特性产生确定性。

Ptides 平台内的驱动器端口作为平台的输出。当它从平台 Ptides 模型接收事件时,该事件有一个逻辑时间戳 t 。驱动器把 t 当作相对于平台时间的截止时间 (deadline)。也就是,发送到驱动器的带有时间戳 t 的事件是一条执行不迟于 (平台)时间 t 的物理行为的命令。因此,驱动器,与传感器一样也进行逻辑时间和物理时间的桥接。

例 10.14 在图 10-18 中的 Ptides 模型中,在 PtidesPlatform 1 中,假设 SensorPort 产生带有时间戳 t_s 的事件。这代表传感器测量的平台时间。进一步假设 Computation 是一个零延时角色,它通过产生具有相同时间戳 t_s 的输出事件对来自 SensorPort 的事件进行响应。因此,TimeDelay 角色的输出将有时间戳 $t_s + d_1$,其中 d_1 是 TimeDelay 角色的延迟。该事件进行到 ActuatorPort,把时间戳 $t_s + d_1$ 作为截止时间。即执行器应该在不迟于平台时间 $t_s + d_1$ 内产生行动。

默认情况下,当执行 Ptides 模型时,假设角色是瞬时执行的 (在平台时间内)。因此,图 10-18 中 ActuatorPort 的截止时间永远不会违反。事实上,在仿真中,ActuatorPort 可能早于平台时间 t_s 执行其动作。这是不现实的,因为这个平台的任何物理实现都将产生某些延

迟。它不能立即对传感器事件做出反应。更实际的仿真模型可以通过将 10.4 节的执行切面与 Ptimes 相结合来构造,但本章并不采用该方法做。相反,这里假设平台的物理实现引入了延迟的某些变化,但截止时间将仍然得到满足。验证这些是可调度性问题。

图 10-18 中驱动器端口执行影响物理设施,进而影响传感器端口。有反馈回路和封闭回路的行为将受到该平台延迟的影响,如果延迟是未知的或可变的,那么系统的整个封闭回路行为将是未知的或可变的,将产生非确定性模型。为了恢复确定性,在截止时间配置 Ptimes 驱动器执行其驱动,而不是在截止时间之前。只要事件在截止时间或之前到达,执行器将能够确定地产生驱动,独立于事件的实际到达时间,因此其独立于执行时间变化。PtimesPlatform1 对传感器事件的响应是一个确定性的驱动器事件(在平台时间和甲骨文时间中)。这使得整个封闭回路系统的行为独立于执行时间的变化(如下所示,网络延时)。为了配置提供这种确定性的 Ptimes 驱动器,将 ActuatorPort 的 `actuateAtEventTimestamp` 参数设置为 `true`。因此, Ptimes 提供了一种机制来隐藏潜在的不确定性和可变性(当不满足截止时间时,直到达到故障阈值),产生确定性封闭回路行为。

如果现在超过故障阈值,将自然出现应该做什么的问题?默认情况下,如果 Ptimes 中的驱动器端口接收到一个带有时间戳 t 的事件,且平台时间也已经超过 t ,那么这个端口将抛出一个异常。该异常指出该事件违反了有关平台具有满足截止时间能力的假设。一个设计良好的模型可以捕获此类异常,例如使用模态模型中的错误转移(见 8.2.3 节)。当然,如何处理这类异常,取决于应用程序。这可能是必要的,例如,为了切换到安全操作但是属于降级模式的操作。或者,可能有必要重启系统的某部分,或切换到备份系统。

模型中的多个 Ptimes 平台可通过网络进行通信。当这样的通信发生时,逻辑时间戳与数据一起传输。与驱动器端口不同,当网络发射器端口可用时,它总是立即产生输出,而不是等待平台时间来匹配时间戳。事件的逻辑时间戳将与事件一起发送到网络接收器端口,然后,产生带有相同时间戳的输出事件。

与驱动器端口一样,网络传输端口将时间戳作为截止时间,如果平台时间超过事件到来时的时间戳值^①,端口将抛出一个异常。

例 10.15 在图 10-18 的 Ptimes 模型中,在 PtimesPlatform 1 中,假设传感器在平台时间 t_s 进行测量,因此网络发射器端口接收到一个带有时间戳 $t_s + d_1$ 的事件。进一步假设在平台时间 t_s 接收到这个事件,因为假定角色的执行时间(默认)为零。因此,图 10-18 中的 Network(网络)角色将收到将值(事件传递到 NetworkTransmitterPort 的值)和逻辑时间戳 $t_s + d_1$ 作为其有效载荷的事件。

在平台时间 t_s , NetworkTransmitterPort 将这个有效载荷送入网络。该网络将产生一些延迟,由图中的 Network 角色模拟,并将在时间 t_2 (PtimesPlatform2 的本地平台时间)到达 PtimesPlatform2。PtimesPlatform2 的 NetworkReceiverPort 将生成一个带有(逻辑)时间戳 $t_s + d_1$ 的输出事件,从有效载荷中提取。在图 10-18 中,这个事件将通过另一个 Computation 角色和 TimeDelay 角色。假设 TimeDelay 角色将时间戳增加了 d_2 , PtimesPlatform2 的 ActuatorPort 将收到一个带有时间戳 $t_s + d_1 + d_2$ 的事件。如果 $t_s + d_1 + d_2 \geq t_2$,那么说明截止时间到了。

① 这个截止时间可能会通过改变传输网络发射器端口参数 `platformDelayBound` 来提前或推后,详见下文详解。

如果 ActuatorPort 中的 `actuateAtEventTimestamp` 参数值为 `true`，并且所有传输在截止时间，那么从 platform 1 的传感器到 platform 2 的驱动器的总延时是确定的，并独立于实际的网络延迟和实际计算时间。在分布式系统中延时固定对 Ptides 模型的说明能力是相当重要的。

与 platform1 的驱动器一样，如果 platform2 的驱动器不能满足截止时间，那么 ActuatorPort 将抛出一个异常。这个异常表明违反了关于执行的一些时间假设，例如，网络没有真正满足一个的网络延时上界的假设。模型应该把这个作为一个错误条件处理，例如，对错误处理的发生，导致模型转换为一个安全的但降级模式的操作。

尽管从 platform1 的传感器到 platform2 的驱动器的端到端延时是确定的，但还不十分清楚这个模型的延时是什么。名义上，延时就是逻辑延迟时间 $d_1 + d_2$ 。然而，执行发生的时间 $t_s + d_1 + d_2$ 是相对于 platform 2 的本地平台时钟。然而，这个时间还取决于 platform 1 的时钟，因为当传感器测量发生时 t_s 是 platform 1 的时间。因此，为了有用，分布式 Ptides 系统要求时钟是同步的（见 10.2 节）。它们不需要是完全的同步的，但是如果它们之间的误差是没有边界的，那么端到端延时就没有界限（在甲骨文时间）。

如果这两个平台的时钟完全同步，相对于这些平台时钟，实际延迟为 $d_1 + d_2$ 。当然，甲骨文时间的延迟，取决于这些时钟相对于甲骨文时间的偏移（见图 10-2）。如果这两个时钟完全按照甲骨文时间的速度推进，那么实际延迟将恰好在甲骨文时间 $d_1 + d_2$ 。因此，如果有足够好的时钟和足够好的时钟同步，Ptides 给出精确和确定性的总体时间行为，以满足这些时钟的精度。

为了确保实现满足规范的时间要求，网络接收器端口还需对时间进行约束。如前面所述，如果网络发射器端口在大于事件时间戳^①的平台时间接收到一个事件，那么它将抛出一个异常。所以如果网络接收器端口接收到一条消息，那么它知道消息传播的平台时间不迟于它接收消息的时间戳。接收器有一个“确保实现满足规范要求”的参数 `networkDelayBound`，这是假定的网络延时的上限。当网络接收器接收到一条消息时，它检查消息上没有超过时间戳上的平台时间，加上 `networkDelayBound`，加上一个引起时钟差异和设备延迟的修正因子，还假设参数指定的范围，如下所述。如果平台时间太大，那么网络接收器知道违反了这些假设之一（虽然不知道是哪一个），它抛出一个异常。虽然这个约束是非常微妙的，但模型的结论相对容易理解。

例 10.16 在图 10-18 中的 Ptides 模型中，沿着从传感器到网络接收器端口的路径，沿着同样路径物理延时不能大于逻辑延时。沿着这条路径的逻辑延时是简单的 d_1 ，`TimeDelay` 角色的参数。物理延时是沿着这条路径的角色执行时间（在仿真中默认为零）与网络延时的和。因此，如果网络使延时大于 d_1 ，那么在该图中的模型将失败（默认情况下，虽然其他的错误处理策略可以使用）。

注意，如果用 DE 指示器代替平台中的 Ptides 指示器，那么行为将明显不同。在这种情况下，从 platform 1 的传感器到 platform 2 的驱动器的延迟将包含实际的网络延时。Ptides 的一个关键特性是网络延时和计算时间是从模型的逻辑时间提取的。逻辑时间成为时间行为的规范，而网络延时和计算时间是系统实现的一部分。Ptides 模型能够确定哪种实现将满足

① 这个事件的截止时间可能会因为网络传输端口的参数 `platformDelayBound` 的改变而变得更早或更晚，正如下文介绍的那样。

这个规范的要求和条件。仿真器允许精细条件下的行为评估，这是很难进行有效分析的评估行为，例如考虑 PTP 时钟同步协议的复杂动态。

10.5.2 Ptides 组件

Ptides 端口。Ptides 平台中的端口代表与环境或网络通信的设备。Ptides 端口可以对设备延时建模，虽然默认这些延时为零。每个 Ptides 端口有一个 `deviceDelay` 和一个 `deviceDelayBound` 参数。`deviceDelay` d 为设备的延迟建模。例如，如果一个传感器在平台时间 t_s 做了一个测量，它将产生一个具有时间戳 t_s 的事件。但直到平台时间 $t_s + d$ 事件才会出现。例如， d 可能代表传感器设备提出一个中断请求和处理器响应这个中断请求所需的平台时间。

补充阅读：安全处理分析

Ptides 平台中角色的执行遵循 DE 语义。角色必须按照时间戳顺序处理事件（除非它们是无记忆的）。在仿真中，确保事件按照时间戳顺序处理是简单的，但当部署 Ptides 模型时，事情变得更加复杂。特别是，待部署的系统无法轻易协调跨平台角色点火的调度，每个平台都必须能够做出自己的调度决策。

考虑图 10-19 中的平台模型。这个例子有一个传感器端口和一个网络接收器端口。假设传感器产生一个带有时间戳 t_s 的事件。如果假设每个传感器按照时间戳顺序产生事件，那么调度器可以立即点火 `Computation1`。假设这个点火产生另一个带有时间戳 t_s 的事件，然后在 `Computation3` 的顶部输入产生一个带有时间戳 $t_s + d_1$ 的事件。什么时候点火 `Computation3` 响应该事件？调度器必须确保没有带有时间戳小于或等于 $t_s + d_1$ 的事件，稍后使得底部输入 `Computation3` 的变得可用。

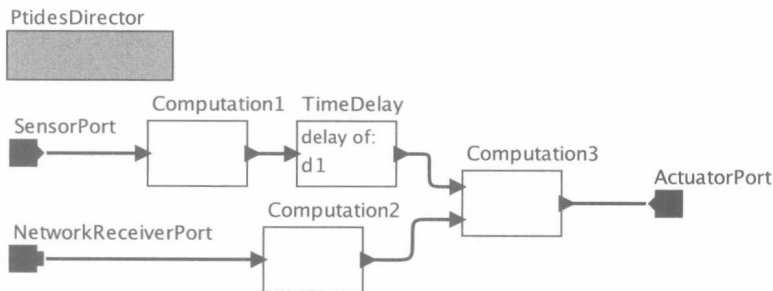


图 10-19 使用简单的 Ptides 例子来说明安全处理 (safe-to-process) 分析

有一种简单的方法，由 Chandy and Misra (1979) 为分布式离散事件仿真开发的，是等待直到 `Computation3` 的底部输入有一个时间戳大于或等于 $t_s + d_1$ 的事件。但这可能导致相当长时间的等待，特别是，如果发生故障，且这条路径上的事件源失效。

由 Jefferson (1985) 提出的另一种方法推理地点火 `Computation3`，假设没有问题的事件将随后到达，如果真的随后到达，通过恢复这个角色的状态反转计算。这种方法在根本上受到无回溯到驱动器的限制。

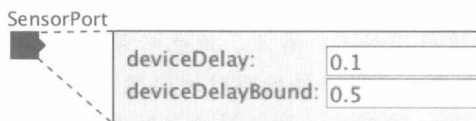
只要满足所有的截止时间，Ptides 方法确保事件按序处理。在本书的例子中，在本

地平台时间达到或超过了 $t_s + d_1$ 时, 可以安全地处理带有时间戳 $t_s + d_1$ 的 Computation3 顶部输入的事件。这是因为可调度性要求带有时间戳 $t_s + d_1$ 或更早的事件在平台时间 $t_s + d_1$ 或更早到达网络接收器端口。

相反, 假设带有时间戳 t_n 的事件在 Computation3 的底部输入。当平台时间满足 $t_n - d_1 + s$ 时, 这个事件是**可安全处理的** (safe to process), 其中 s 是**传感器延时的界限**, 传感器延迟是传感器事件时间戳与对事件对调度器可见之间的时间。第 10 章补充阅读: Ptides 的背景对此进行详细解释。

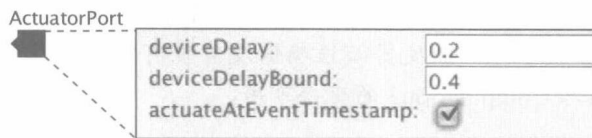
deviceDelayBound d_b 给出了 deviceDelay d 的上界。Ptides 框架假设 deviceDelay 可以在执行期间变化但永远不会超过 deviceDelayBound, 它是不变的。这个上界用于确保**安全处理** (safe to process) 分析 (见第 10 章补充阅读: 安全处理分析), 它确保事件按照时间戳顺序处理。

传感器。传感器端口是特殊的 Ptides 端口, 如下图所示。



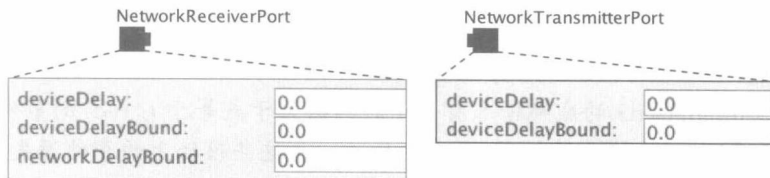
它从环境中接收输入并创建时间戳等于当前平台时间 (这是 PtidesDirector 的当前本地时间) 的新事件, 然后将这一事件传送到事件队列中。传感器在平台时间 t_s 接收的事件在平台时间 $t_s + d$ 产生带有逻辑时间戳 t_s 的事件。

驱动器。执行器端口是一种特殊的 Ptides 端口, 如下图所示:



默认情况下, 当事件的时间戳等于当前平台时间时, 驱动器端口在平台的输出产生事件。如果将 actuateAtEventtimestamp 更改为 false, 那么如果它早些可用可能早些产生事件。deviceDelayBound 参数指定设备的建立时间 (setup time)。具体来说, 将带有时间戳 t 的事件传送给驱动器的截止时间是平台时间 $t - d_b$, 其中 d_b 是 deviceDelayBound 的值。如果不满足截止时间, 将抛出一个异常。

网络发射器和接收器。网络发射器端口和网络接收器端口也是特殊的 Ptides 端口, 如下图所示。



NetworkTransmitterPort 从 Ptides 平台内部获得事件, 并给外部发送一个编码事件时间戳和值 (有效载荷) 的记录。NetworkReceiverPort 提取时间戳和有效载荷, 并在目的 Ptides 平台内产生具有指定值和时间戳的事件。

参数 `networkDelayBound` (d_N) 指定在输入令牌到达接收器前在它网络中花费的假定最大时间。它用来确定事件是否可以安全处理, 或者另一个带有较小时间戳的事件是否仍然在网络中。如果实际的网络延时大于这个界限同时延迟导致太晚接收到消息, 那么网络接收器端口将抛出异常。

10.6 小结

对复杂的计时系统建模是一项艰难的工作。人们对时间的统一构造往往持有一种初级的概念, 认为它是被物理世界中所有的参与者共享的。但这样的认知是不准确的, 时间度量上的误差以及时间在逻辑与物理概念上的不同都会对实际的系统造成强烈影响。Ptolemy 工程最近研究工作的一个主要焦点是为现实中实际上非固定时间的系统提供一种固定的功能模型。

非常感谢 Yishai Feldman 和 Stavros Tripakis 为本章提供的有益建议。

Ptera: 面向事件的计算模型

Thomas Huining Feng 和 Edward A. Lee

在第 6、7 章概述的 FSM 和 DE 模型主要关注事件以及这些事件之间的因果关系。原子事件从概念理解就是事件发生在某个瞬间。一个**面向事件的模型** (event-oriented model) 定义了一段时间内发生的事件的集合。具体地说, 它包含一系列事件, 一般按时间顺序排列, 并定义如何由这些事件点火其他事件。如果有外部提供的事件, 它也定义这些事件如何点火外部的事件。在 DE 域中, 事件的时间由计时源和延迟角色控制 (见第 7 章补充阅读: 时钟角色和时间延迟)。FSM 域中的模型主要响应外部提供的事件, 但也可能在条件中使用 timeout 函数产生计时事件 (详见表 6-2)。

有许多方法可以用于说明面向事件的模型 (详见第 11 章补充阅读: 面向事件模型的代表法和语言)。本章描述一种新颖的模型, 称为 **Ptera** (for Ptolemy event relationship actors), 首先由 Feng et al. (2010 年) 提出。Ptera 旨在与其他 Ptolemy II 计算模型更好地交互操作, 提供模型层次结构并以确定性的方式处理并发性问题。

11.1 扁平模型的语法和语义

一个扁平 (即, 非分层的) Ptera 模型是一个通过有向边连接顶点的图, 如图 11-1 所示, 图中包含两个顶点和一条边。一个顶点相当于一个**事件** (event), 一条有向边表示一个事件触发另一个事件发生的条件。如本章后文所述, 可以设定顶点和有向边的属性和参数的范围。

在分层 Ptera 模型中, 顶点也可以表示子模型, 这个子模型可能是其他的 Ptera 模型、FSM、使用其他 Ptolemy II 指示器的角色模型, 甚至定制的 Java 代码。唯一的要求是, 它们的行为必须符合角色抽象语义 (actor abstract semantics) 的定义, 解释如下。

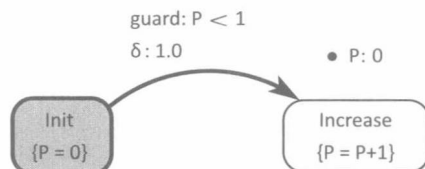


图 11-1 包含两事件的简单 Ptera 模型

补充阅读：面向事件模型的代表方法和语言

已经有很多方法和语言来描述面向事件的模型。目前广泛应用的是 **UML 活动图** (activity diagram), 它是诞生于 20 世纪 60 年代的经典**流程图** (flowchart) 的衍生物 (见 http://en.wikipedia.org/wiki/Activity_diagram)。在活动图中, 一个圆角矩形表示一个**活动**, 从一个活动指向另一个活动的箭头表示两个活动之间的因果关系。菱形图表示点火一个分支活动的检测条件。活动图包括分叉 (split) 和汇合 (join) 活动, 产生多个并发活动, 并等待这些活动完成。虽然 UML 符号是常见的, 但其并发性的语义 (甚至是活动) 并不明确。活动可以理解为用户事件, 在这种情况下, 它们是原子的, 或者它们可

能需要消耗时间,在这种情况下,一个活动的触发和完成就是事件。当并发活动的连接被添加或删改时,活动图的含义就会变得模糊,并且时间模型也没有定义。然而,活动图往往直观易理解,因此被证明是一种与面向事件模型通信的有效方法。

另一相关的表示方法是**业务流程模型及符号**(Business Process Model and Notation, BPNL),与 UML 一样,目前由 OMG 维护和开发。BPNL 区分事件和活动,允许两者在同一个图中。它还提供了一种分层结构和并发性的形式,其具有复杂交互和同步机制。

第三个相关的表示方法是**控制流图**(Control Flow Graph, CFG),由 Allen (1970) 提出,目前广泛应用于编译器优化、程序分析工具和电子设计自动化。在 CFG 中,图中的节点表示程序的**基本块**(basic block),它是没有任何分支或流控制结构的指令序列。这些基本块被看作原子,因此可以认为是事件。基本块之间的连接表示一个程序运行时可能遵循的流控制序列。

补充阅读: Ptera 的背景

Ptera 来源于**事件图**(event graph),由 Schruben (1983) 提出。事件图中的块(称为顶点)包含事件,和事件处理时要执行的动作。事件之间的连接(称为“有向边”)表示可以通过布尔和时间表达式加以判定的调度关系。事件图是时控的,时间延迟与调度关系有关。尽管没有具体的可视化表示形式,但是每个事件图都有一个事件队列。在多线程执行时,原则上可以使用多个事件队列。在每一步的执行中,执行引擎从事件队列中删除下一个事件并进行其处理。接着执行事件的相关动作,将调度关系指定的额外事件插入事件队列中。

原始事件图不支持层次结构。Schruben (1995) 提出两种支持层次结构的方法。一种方法是将子模型与调度关系相关联,其中子模型的输出是一个表示调度关系延迟的数字。另一种方法是将子模型与事件相关联,而不是与调度关系(Som and Sargent, 1989)。处理此类事件首先将调度子模型中唯一的起始事件,进而可能进一步调度子模型中的事件。当处理完一个预定结束事件时,其子模型的执行终止,与该子模型相关联的事件被认为正在处理。Buss and Sanchez (2002) 支持层次结构的第三种方法,引入监听模式作为额外连接机制来组成事件图。

Ptera 基于事件图,但对其进行扩展后支持异构分层建模。Ptera 模型的组成形成一个层次结构模型,可扁平化得到一个没有层次结构的等效模型。Ptera 模型符合角色抽象语义,允许它们包含或者被其他类型的模型包含,因此使层次化异构设计得以实现。Ptera 模型可以与 Ptolemy II 中的其他计算模型自由组合。

Ptera 模型包含带参数的外部可见接口、输入端口和输出端口。改变参数和到达输入端口的数据可以触发 Ptera 模型内的事件,在这种情况下,接口变成了角色。另外,事件动作可以通过程序员遵循协议,对采用命令式语言(Java 或 C)编写的程序进行定制。

11.1.1 入门实例

例 11.1 一个 Ptera 模型的例子如图 11-1 所示。这个模型包含两个顶点(事件), Init 和 Increase 以及一个变量 P (初值为 0)。当模型执行时,用新的值更新该变量。Init 是**初始事**

件 (initial event)(初始参数设置为 true)，图中用边框加粗的圆角矩阵来表示。

在开始执行时，所有初始事件预计发生在模型时间 0。(如后面所讨论的，即使理论上多个事件发生在同一时间，但这些事件仍有一个定义明确的执行顺序。) 该模型的“事件队列”保存预定事件实例的列表。当模型时间到达事件预计发生的时间时 (在该事件的时间戳)，从事件队列中移除一个事件并进行其处理。

Ptera 事件可能与具体动作有关，显示在顶点上的大括号内。

例 11.2 在图 11-1 中，例如，Init 指定动作“ $P=0$ ”，即当 Init 被执行时将 P 设置为 0。从 Init 事件到 Increase 事件的边 (连接) 称为调度关系 (scheduling relation)。它用布尔表达式“ $P<1$ ”判定 (这意味着当满足这个条件时，转移才能发生) 判定，延迟为 1.0 时间单位 (由符号 δ 表示)。Init 事件处理后，如果 P 的值小于 1 (因为 P 的初始值为 0，所以满足这个条件)，那么将在时间 1.0 执行 Increase 事件。当在时间 1.0 执行 Increase 事件时，执行动作“ $P=P+1$ ”， P 的值增加为 1。

Increase 事件执行结束后，事件队列为空。因为没有安排更多的事件，所以执行结束。

在这个简单的例子中，在任何时刻事件队列中至多有一个事件 (Init 事件或 Increase 事件)。但事件队列中可用于调度的事件数量不受限制。

例 11.3 图 11-2 表示一个稍微复杂的 Ptera 模型，它要求事件队列的规模大于 1。在这个模型中，Init 事件设定 IncreaseA 事件延迟 1 个时间单位执行，IncreaseB 事件延迟 2.0 个时间单位执行。从 Init 事件开始的两个调度关系的判定式默认值为“true”，因此没有进行可视化表示。当执行 IncreaseA 时，把变量 A 的值增加 1，并重新调度此事件，在事件队列中创建另一个事件实例，循环直到 A 的值等于 10。(从 A 到自身的调度关系的模型时间延迟 δ 是隐藏的，因为它的默认值为“0.0”，这意味着在当前模型时间调度该事件，而不是下一个微步) 同样，在当前模型时间 IncreaseB 反复增加变量 B ，直到 B 的值等于 10。

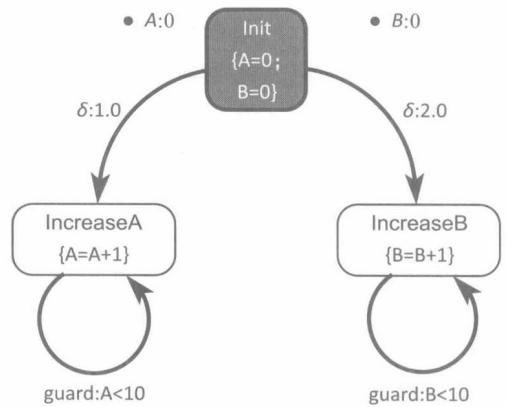


图 11-2 事件队列具有多个事件中的模型

11.1.2 事件参数

与 C 语言中的函数一样，一个事件可能包含一个“形参”列表，其中给每个参数设定了名称和类型。

例 11.4 图 11-3 对图 11-2 进行了修改，在 IncreaseA 事件和 IncreaseB 事件中添加了整型参数 k 。传入关系确定这些参数的值，并确定变量 A 和 B 的增量。(图中的虚线边表示一个取消关系，这将在下一小节讨论。)

指向带参数事件的每个调度关系必须指定其“参数”属性中的表达式列表。当处理事件时，

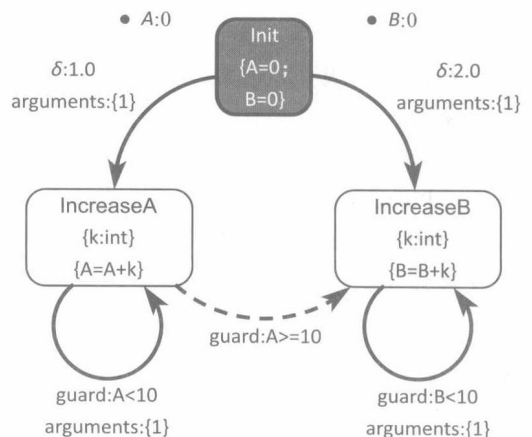


图 11-3 带参数的事件转换条件的模型

这些表达式用来确定参数的实际值。在这个例子中, 指向 IncreaseA 事件和 IncreaseB 事件的所有调度关系都在其参数属性中指定了 “{1}”, 这意味着当执行这些事件时, 参数 k 的值应为 1。事件的动作、条件和延迟都可以调用这些参数值。

11.1.3 取消关系

取消关系 (canceling relation) 用事件之间的虚线边来表示, 由布尔表达式确定转移条件, 并且它不能有延迟和参数。当某事件正在执行且具有取消关系时, 若转移条件为 true 且目标事件已经在事件队列中, 则将目标事件从事件队列中移除而不被执行。换句话说, 取消关系取消了安排的事件。若目标事件被多次调度, 即有多个事件实例在事件队列中, 则此取消关系只移除了第一个事件实例。若目标事件没有调度, 则取消关系无效。

例 11.5 图 11-3 是一个取消关系的例子。在时间 1.0 执行最后一个 IncreaseA 事件导致 IncreaseB 事件 (由 Init 事件设定在时间 2.0 执行) 被取消。因此, 变量 B 不会增加。

应该注意的是, 取消关系并不会丰富模型内容。事实上, 带有取消关系的模型总是可以转换为一个没有取消关系的模型, 与 Ingalls et al. (1996) 提出模型的一致。尽管如此, 取消关系的存在可以产生结构更紧凑且更易理解的模型。

11.1.4 同时事件

同时事件 (simultaneous event) 是指安排在相同模型时间执行的事件队列中的多个事件实例。

例 11.6 例如, 在图 11-3 中, 如果将两个 δ 都设置为 1.0, 那么模型如图 11-4 所示。由 Init 调度的 IncreaseA 事件和 IncreaseB 事件的实例变成同时事件。此外, 尽管 IncreaseA 事件的多个实例在相同模型时间执行, 但它们在超密时间的不同微步出现, 并且它们不在事件队列中共存, 所以 IncreaseA 的多个实例不是同时事件。

可以说, 这是一个检测同时事件的模型检测问题 (Clarke et al., 2000)。

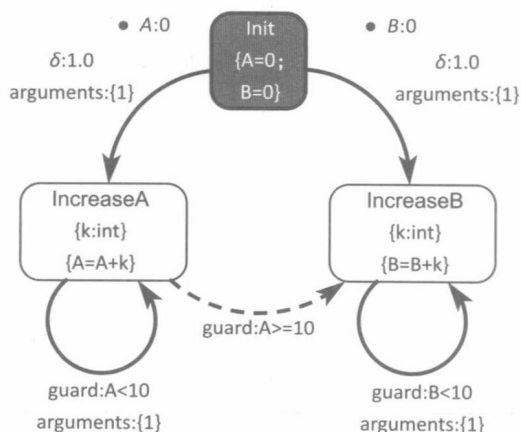


图 11-4 具有同时事件的模型

11.1.5 潜在的非确定性

当存在同时事件时, 事件处理的不明确的顺序引入潜在的非确定性问题。

例 11.7 例如, 在图 11-3 中, 变量 A 和 B 的最终值是多少? 假设 IncreaseA 的所有实例在 IncreaseB 的所有实例之前执行。在这种情况下, B 的最终值是 0。反过来, 如果 IncreaseB 的所有实例在 IncreaseA 的所有实例之前执行, 那么 B 的最终值是 10。

表 11-1 显示了 4 种可能的执行路径, 似乎与模型的定义一致。列按照事件执行的顺序从左到右排列。这些路径包括如下 4 种情况: IncreaseA 的所有实例在 IncreaseB 的所有实例之前执行; IncreaseB 的所有实例在 IncreaseA 的所有实例之前执行; IncreaseA 和 IncreaseB 以两种不同方式交替执行。还有很多其他可能的执行路径。

不同的执行路径导致 A 和 B 的最终值不同。IncreaseA 的最后实例, 即将 A 增加到 10,

总是取消事件队列中的下一个 IncreaseB（如果还有）。IncreaseB 的实例总共有 10 个，没有定义明确的执行顺序，被取消的事件将是其中的任意一个。

为了避免这些不确定性的执行结果，将采用后面讨论的策略。

表 11-1 图 11-4 中模型的四种可能的执行路径

1) IncreaseA 总是在 IncreaseB 之前调度：						
Time	0.0	1.0	1.0	...	1.0	
Event	Init	IncreaseA	IncreaseA	...	IncreaseA	
A	0	1	2	...	10	
B	0	0	0	...	0	
2) Increase B 总是在Increase A之前调度：						
Time	0.0	1.0	1.0	...	1.0	1.0
Event	Init	IncreaseB	IncreaseB	...	IncreaseB	IncreaseA
A	0	0	0	...	0	1
B	0	1	2	...	10	10
Time	1.0	...	1.0			
Event	IncreaseA	...	IncreaseA			
A	2	...	10			
B	10	...	10			
3) IncreaseA 和 IncreaseB 交替出现，由 IncreaseA 先开始：						
Time	0.0	1.0	1.0	1.0	1.0	...
Event	Init	IncreaseA	IncreaseB	IncreaseA	IncreaseB	...
A	0	1	1	2	2	...
B	0	0	1	1	2	...
Time	1.0	...	1.0	1.0		
Event	IncreaseA	IncreaseB	IncreaseA			
A	9	9	10			
B	8	9	9			
4) IncreaseA 和 IncreaseB 交替出现，由 IncreaseB 先开始：						
Time	0.0	1.0	1.0	1.0	1.0	...
Event	Init	IncreaseB	IncreaseA	IncreaseB	IncreaseA	...
A	0	0	1	1	2	...
B	0	1	1	2	2	...
Time	1.0	...	1.0	1.0	1.0	
Event	IncreaseB	IncreaseA	IncreaseB	IncreaseA		
A	8	9	9	10		
B	9	9	10	10		

11.1.6 LIFO 和 FIFO 策略

Ptera 模型指定了一个后进先出（Last In, First Out, LIFO）或先进先出（First In, First Out, FIFO）策略来控制事件队列中事件实例的访问顺序，以确保具有确定性的结果。采用 LIFO（默认）策略，后安排的事件先处理，FIFO 策略正与此相反。Ptera 模型中的 LIFO（默认值为 true）参数决定了对 LIFO 或 FIFO 的选择。

如果采用 LIFO 策略来执行图 11-4 中的模型，则表 11-1 中的执行路径 3 和 4 不可能出现。这产生两种可能的执行路径，取决于 IncreaseA 与 IncreaseB 哪个先执行（选择结果将取决于稍后讨论的调度规则）。

例 11.8 假设 IncreaseA 事件先执行。根据 LIFO 策略，IncreaseA 事件的第二个实例将先于 IncreaseB 事件执行，其中第二个实例的执行由第一个实例调度，IncreaseB 事件由 Init 事件调度。第二个实例再次调度下一个实例。采用这种方式，继续执行 IncreaseA 事件的实

例, 直到 A 的值等于 10, 当取消 IncreaseB。这样产生了执行路径 1。

相反, 如果先执行 IncreaseB, IncreaseB 事件的所有 10 个实例在 IncreaseA 之前执行。这会产生执行路径 2。

然而, 采用 FIFO 策略, IncreaseA 和 IncreaseB 的实例交错执行, 产生表中的执行路径 3 和 4。

在实际中, LIFO 应用更普遍, 因为它执行事件链。前一事件与后一事件之间的调度没有延迟。该方法是一个原子操作, 在某种意义上事件链中正在处理的事件不会被该链中另一个事件干扰。这种方式很方便指定某些工作流, 这些任务需无干扰且有序地完成。

11.1.7 优先级

当相同事件以相同延迟 δ 调度多个事件时, 可以通过分配给调度关系的优先级 (priority) 数确定事件执行的顺序。优先级是一个默认值为 0 的整数 (也可以为负数)。

例 11.9 图 11-5 中的同时事件 E1 和 E2 分别由调度关系 r1 和 r2 调度 (延迟都是 0.0, 因为 δ 未赋值)。若 r1 的优先级比 r2 的高 (即优先级数较小), 则事件 E1 先于事件 E2 执行, 反之亦然。

例 11.10 在图 11-4 中, 如果 Init 到 IncreaseA 的调度关系的优先级数是 -1, 从 Init 到 IncreaseB 的调度关系的优先级数是 0, 则 IncreaseA 的第一个实例先于 IncreaseB 执行。表 11-1 中执行路径 2 和 4 是不可能的。另一方面, 如果从 Init 到 IncreaseA 的调度关系的优先级数是 1, 则 IncreaseB 的第一个实例先执行。表 11-1 中的执行路径 1 和 3 是不可能的。

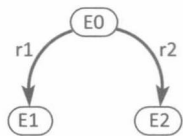


图 11-5 事件 E0 在相同延迟后调度 E1 和 E2 的情况

11.1.8 事件命名及调度关系

Ptolemy II 的 Ptera 模型中的每个事件和每个调度关系都有一个名称。例如, 在图 11-4 中, Init、IncreaseA 和 IncreaseB 都是事件的名称。这些名称可以通过编辑器设置 (见图 2-15)。Vergil 给事件分配一个默认名称, 在分层模型的每一层, 这些名称是唯一的。当同时事件调度关系的优先级相同时, Ptera 模型用这些名称来确定同时事件的执行顺序。

在图 11-5 中, 若调度关系 r1 和 r2 有相同的延迟 δ 和相同的优先级, 那么根据名称确定事件执行的顺序[⊖]。事件 E1 和 E2 的执行顺序首先从比较事件的名称来确定。在扁平模型中, 这些名称是不同的, 所以它们有效的字母排列顺序。字母排列靠前的将先执行。

在分层模型 (下面将讨论) 中, 同时事件可能有相同的名称。在这种情况下, 将使用调度关系的名称来确定。虽然这些通常不会显式地表示出来, 但是可以通过鼠标悬停在调度关系上查看。

11.1.9 原子性设计

在某些应用程序中, 设计者需要确保在其他同时事件存在时, 事件序列的原子性执行。也就是说, 整个事件序列的执行应该先于任何其他同时事件。图 11-6 中的两种设计模式都

[⊖] 在 Ptera 中, 不论非确定性地选择或并发执行两个事件, 都是有效的, 但是这将导致非确定性问题, 所以当前实现中, 我们选择定义执行顺序。

可以用来保证原子性设计，而无需设计人员明确地进行关键部分的控制（如在命令式编程语言的情况下）。

图 11-6a 中的设计模式用于顺序和自动地执行任务，假设选择 LIFO 策略。即使在模型中还存在其他事件（图中未画出），那些事件也不能与任务交错执行。因此，任务之间的中间状态不受其他事件的影响。

最终事件（final event）END，是一类特殊的事件，它删除事件队列中的所有事件。最后事件用于强制终止，即使事件队列中还有剩余事件。用双线边框填充的顶点表示。

在图 11-6b 中的设计模式用于进行任务的执行直到满足条件 G 。这种模式再次使用 LIFO 策略。当起始事件（Start）执行之后，调度所有任务。在这种情况下，依照字母顺序排序，第一个执行的是 Task1。Task1 之后，若 G 为 true，接下来执行 END，它终止任务的执行。若 G 不是 true，则执行 Task2。任务继续执行，直到这时 G 变为 true 或者所有任务都执行完成但是 G 还是 false。

11.1.10 面向应用的实例

本节将介绍两种简单的 Ptera 模型，它们的特性与很多重要的系统类似。首先介绍一个简单的多服务、单队列系统：洗车系统。

例 11.11 在该例中，多个洗车机共享一个队列。当一辆汽车达到时，它停在队尾等待服务。洗车机以“先来先服务”的原则每次为队列中一辆汽车提供服务。汽车到达的时间间隔和服务时间由随机程序生成，即边上的调度延迟。

图 11-7 中的模型分析可用服务器的数量和过去一段时间等待的汽车数量。servers 变量的初始值是 3，表示服务器的总数量。Queue 变量从 0 开始表示在开始时队列中没有汽车正在等待。Run 是一个初始事件。在第三个变量 SimulationTime 所定义的时间到达时，将调度 Terminate（终止）最终事件，终止任务执行。

Run 事件也调度 Enter 事件的第一个实例，使得第一辆车在延迟 “ $3.0+5.0*\text{random}()$ ” 之后到达，其中 $\text{random}()$ 函数返回一个在 $[0, 1)$ 内均匀分布的随机数。当 Enter 事件发生时，使队列大小的变量 Queue 增加 1。Enter 事件可以对自身进行调度。若有可用的服务器，也可以调用 Start 事件。LIFO 策略确保了 Enter 事件和 Start 事件可以自动执行，因此当经过 Enter 和 Start 调度关系的条件判定后，服务器的值不会被队列中的另一事件改变。换句话说，当一辆车进入了队列且已经开始清洗时，这台洗车机不会为另一辆车提供服务。

通过减少可用服务器的数量和队列中车辆的数目 Start 事件仿真洗车。服务时间是 “ $5.0 + 20.0 * \text{random}()$ ”。这段时间之后，Leave 事件发生，它表示对这辆车的服务结束。当一辆车离开，可用服务器的数量一定大于 0（因为这时至少有一台服务器可用），所以若队列中至少有一辆车，则 Leave 事件立刻调度 Start 事件。由于 LIFO 策略提供的原子性，所以

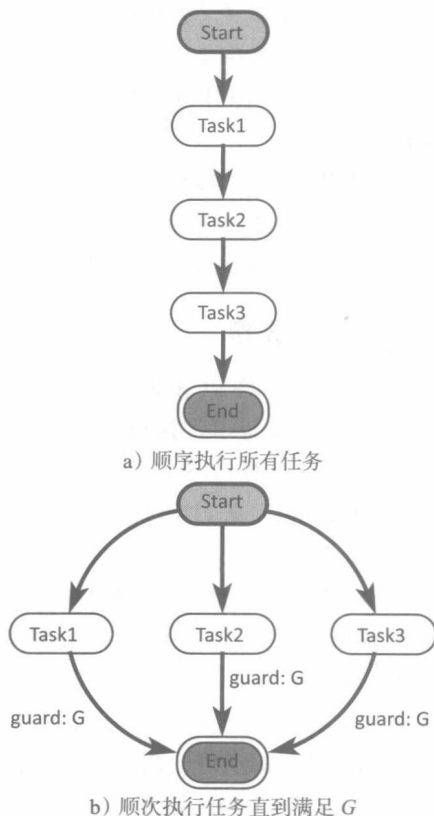


图 11-6 控制任务的两种设计模式

模型将检测队列的大小并在随后的 Start 事件中减小该值，不允许队列中的其他任何事件干扰。

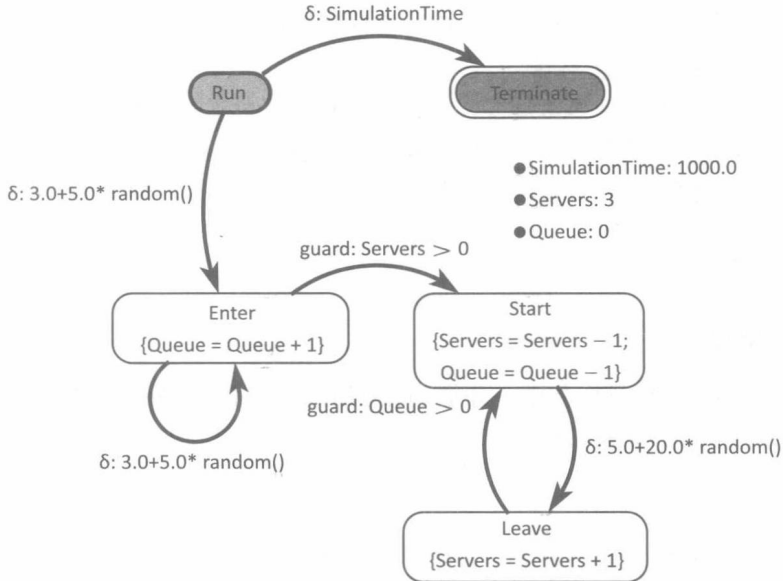


图 11-7 仿真洗车系统的模型

Terminate 事件终止执行，是执行开始时预先定义的事件。没有该事件，模型的执行将不会终止，因为事件队列可能从不为空。

补充阅读：模型执行算法

下面定义扁平模型语义的执行算法。在算法中，符号 Q 表示事件队列。当 Q 为空时，算法终止。

1. 初始化 Q 为空
2. 对于每个符合 $\leq e$ 顺序的初始事件 e

- 1) 创建实例 i_e
- 2) 设置 i_e 的时间戳为 0
- 3) 将 i_e 添加到 Q 中

3. 当 Q 不为空时

- (a) 从 Q 中移除事件 e 的第一个实例 i_e
- (b) 执行 e 的动作
- (c) 若 e 是最终事件，则终止
- (d) 对于每个 e 的取消关系 c

从 Q 中移除 c 指向的事件的第一个实例，若存在

- (e) 令 R 表示 e 的调度关系的列表
- (f) 根据延迟、优先级、目标事件 ID、按重要性排序的调度关系的 ID 对 R 进行分类
- (g) 创建新的空队列 Q'
- (h) 对于 R 中的每个条件为 true 的调度关系 r

- 1) 评估 r 指向的事件 e' 的参数
- 2) 创建 e' 的实例 $i_{e'}$, 并将它与参数关联
- 3) 设置 $i_{e'}$ 的时间戳大于有 r 延迟的当前模型时间
- 4) 将 $i_{e'}$ 添加到 Q' 队尾
- (i) 通过合并 Q' 和 Q 来创建 Q'' , 并保存 Q' 和 Q 原始的事件顺序。对任意 $i' \in Q'$ 和 $i \in Q$, 当且仅当采用 LIFO 策略时在 Q'' 中 i' 先于 i 出现, 且 i' 的时间戳不大于 i 的时间戳, 若采用 FIFO 策略, 则 i' 的时间戳严格小于 i 的时间戳。
- (j) 将队列 Q'' 赋值给 Q

11.2 层次模型

层次结构可以降低模型复杂度, 并提高模型的可重用性。分层复合建模 (hierarchical multimodeling) 使多个计算模型可以结合起来, 每个模型拥有自己的层次。这里介绍如何构建层次 Ptera 模型。下一节将介绍如何分层地将 Ptera 模型与其他计算模型相结合, 如前面章节所述。

例 11.12 考虑有两个站点的洗车系统: 一个站点有 1 个服务器, 另一个站点有 3 个服务器, 每个站点都有自己的队列。图 11-8 显示了图 11-7 的分层修改图, 它有两个站点。它的顶层模拟一个执行环境, 其中 Run 事件是唯一的初始事件, Terminate 事件为最终事件, Simulate 事件与子模型相关联。子模型建立了具有给定服务器数的洗车系统模型。

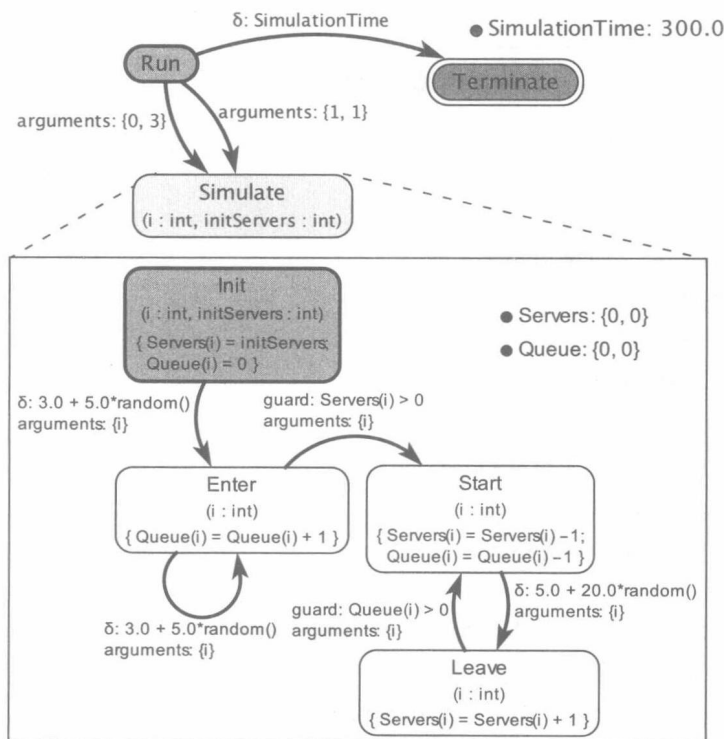


图 11-8 通过两个设置来模拟洗车系统的分层模型

指向 Simulate 事件的两调度关系导致子模型触发其本地事件队列中的 Init 事件的两个实例。这表示两个并发模拟的开始, 一个有 3 个服务器 (通过左边调度关系的第二个参数表示), 另一个有 1 个服务器。初始化调度关系的优先级并未明确规定。因为两个模拟相互独立, 所以它们开始的顺序并没有显著的影响。事实上, 这两个模拟甚至可以同时发生。

参数 i (Init 事件的两个调度关系的第一个参数) 用来区分这两个模拟。与图 11-7 中的模型相比, 子模型中的 Servers 变量扩展为具有两个元素的数组。Servers (0) 表示模拟 $i=0$ 的服务器的数量, Servers (1) 表示模拟 $i=1$ 。变量 Queue 以同样的方式扩展。子模型中的每个事件也需要一个参数 i (指定模拟序号), 并将其发送给调度的下一个事件。这确保在一个模拟下, 即使共享相同的模型结构事件和变量都不会受到其他模拟环境的影响。

理论上, 可以通过初始化一个子模型多次来多次执行它的实例。然而事件队列和变量不能复制, 因此变量必须定义成数组, 并将额外的索引参数 (本例中的 i) 提供给每个事件。

面向角色类 (actor-oriented class) (见 2.6 节) 提供了另一种方法来创建一个子模型的多个可执行实例。子模型可以定义为并行执行多个实例的类。

11.3 异构组合

Ptera 模型可以由其他计算模型来组成。本节将介绍这种组成方式的例子。

11.3.1 Ptera 与 DE 组合

与 Ptera 模型一样, 离散事件 (DE) 模型 (见第 7 章) 是基于事件的。但是 DE 模型中的表示方法存在很大差异。在 DE 中, 模型的组件, 角色, 消耗输入事件并产生输出事件。在 Ptera 中, 完整的 Ptera 模型通过产生输出事件对输入事件进行响应, 所以 Ptera 子模型在 DE 模型中产生自然角色。事实上, DE 和 Ptera 的组合是可以产生不同核心内容的层次结构模型。

例 11.13 在图 11-8 中, 汽车到达模型与汽车服务模型同时存在, 在模型中并不能轻易地将两者分开。如果想要改变模型, 比如使得汽车到达模型服从泊松过程, 应该怎么办? 图 11-9 中的模型, 在顶层设计中采用了 DE 指示器, 并将汽车到达模型与汽车服务模型分开。这个模型与图 11-8 中的模型具有相同的行为, 但是它比较容易用 PoissonClock 角色替换 CarGenerator。

在图 11-9 中, CarGenerator 中的 Init 事件在一个随机延迟后调度第一个 Arrive 事件。每个 Arrive 事件调度下一个事件。当执行此事件时, Arrive 事件产生一个汽车到达信号并使用赋值 “output = 1” (其中 output 是输出端口的名称) 通过输出端口发送该信号。在本例中, 分配给输出端口的值 1 并不重要, 因为本书感兴趣的是产生输出事件的时间。

图 11-9 中也显示了服务器的内部设计。与前文中的洗车模型类似, 不同之处在于有一个额外的接收 DE 事件的 carInput 端口, 此事件表示外部汽车的到达信号; 通过该端口处理输入的 Enter 事件。在 Servers 组件中, 并未假设汽车到达源。在顶层中, CarGenerator 的输出端口与 Servers 的输入端口之间的连接显示地表示生产者消费者关系, 并产生更多模块化的和可重用的设计。

TimedPlotter 显示可用的服务数量和随着时间的推移在队列中等待的汽车数量。在图中的等待队列的图示可知, 随着时间的推移有 5 辆车在等待。

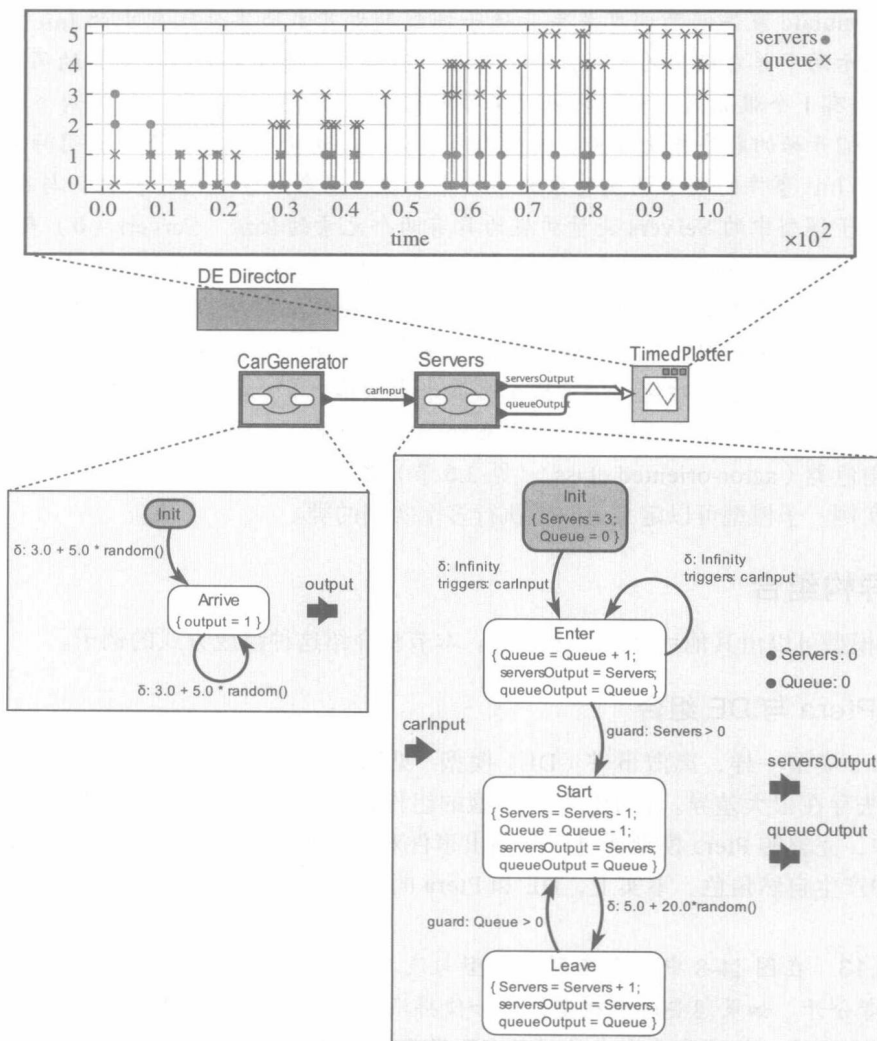


图 11-9 在分层结构中采用 DE 和 Ptera 的洗车模型

在 DE 模型中的 Ptera 模型将在如下情况下执行，接收到 DE 模型输出的输入事件，或一个调度关系超时，即 δ 失效。

例 11.14 在图 11-9 中的 Servers 模型中，从 Init 事件到 Enter 事件的关系标记为 $\delta: \text{Infinity}$ ，这意味着超时永远不会无效。也标记了 `triggers: carInput`，这意味当 `carInput` 端口上的 DE 模型输出的事件到达时，将执行 Enter 事件。

调度关系可以用 `triggers` 属性标记，它指定端口的名称，以逗号分隔。它用于调度 Ptera 子模型中事件来对外部输入做出响应。该属性与延迟 δ 结合来确定事件执行的时间。假设在模型中，触发器参数是“ p_1, p_2, \dots, p_n ”。当模型时间是 δ 时，执行事件，其中 δ 大于满足调度关系的时间，或在 p_1, p_2, \dots, p_n 端口的任一端口接收一个或多个 DE 事件的时间。为了使调度无限期等待输入的事件，给参数 δ 赋值 `Infinity`。

为了测试一个端口是否有输入，访问一个专有的布尔变量，其名为端口名后加上字母串“`_isPresent`”，这与有限状态机 (FSM) 类似。为了引用端口上的可用输入值，端口名可以用于表达式中。

例 11.15 调度图 11-9 中的 Enter 事件尤其有限地等待 carInput 端口的 DE 事件。当接收到一个事件, Enter 事件将被先于它的调度时间执行, 且队列大小加 1。在特定情况下, 输入值可以忽略不计。

经由输出端口发送 DE 事件, 任务可以在事件的执行命令中的时候写入, 该事件的端口名可以写在事件左边, 赋值表达式可以写在事件右边。输出时间戳与事件执行模型时间通常相等。

11.3.2 Ptera 与有限状态机组组合

Ptera 模型也可以与不计时间模型 (如有限状态机 FSMS) 组成。Ptera 模型包含与事件相关联的有限状态机子模型时, 在事件执行或输入端口接收到输入时, 它将点火有限状态机。第 6 章详述了有限状态机。

例 11.16 为了说明 Ptera 和 FSM 的组成, 考虑下述情况。若等候的汽车很多, 司机也许会避免将车进入队列。这可能会导致较低的到达率 (或较长的平均间隔时间)。相反, 如果在队列中有相对较少的汽车, 司机总是将车进入队列, 导致较高的到达率。

针对该场景的改进模型如图 11-10 所示, 在顶层中, 将服务器 Servers 的 queueOutput 端口 (其内部设计与图 11-9 相同) 反馈给 CarGenerator 的 queueInput 端口。图 11-10 中的有限状态机子模型修改 CarGenerator 中的 Update 事件, 它继承了其上层的端口, 允许其进行 queueInput 端口接收输入的转移条件检测。一般情况下, 在 FSM 子模型中的活动也可以通过输出端口产生数据。

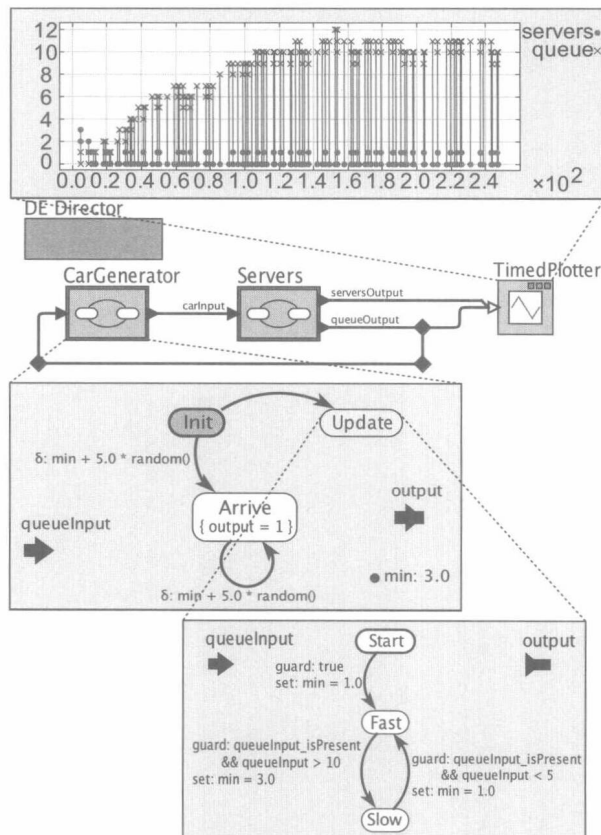


图 11-10 使用分层组成中的 DE、Ptera 和 FSM 的洗车模型

当执行 CarGenerator 中的 Update 事件时，将有限状态机子模型设置为其初始状态。当其第一次点火后，有限状态机进入 Fast 状态，并将最小间隔时间设置为 1.0。随后，使用表达式 $1.0 + 5.0 * \text{random}()$ 生成间隔时间。注意，min 变量是在 CarGenerator 中定义，作用域规则使得上层的有限状态机可以读写该变量。

当 Ptera 模型在端口接收到输入时，点火所有已初始化的子模型，不管这些子模型使用的是何种计算模型。

逆向组成 Ptera 子模型由有限状态机中的细化组成，也很有趣。通过改变状态，子模型可以禁用和启用，通过模式切换执行。这种类型的组成是在第 8 章中介绍的由模型提供。

11.4 小结

Ptera 模型提供了一种替代 FSM 和 DE 模型的方法，提供对基于事件系统建模的互补方法。Ptera 模型与其他的模型的文体风格不同。与有限状态机 (FSM) 中的状态和 DE 模型中的角色相比，Ptera 模型中的组件是事件。连接事件的调度关系表示因果关系，其中一事件可在特定条件（条件表达式、超时和输入事件）下触发另一事件。

建模的基础结构

本书第三部分侧重于介绍 Ptolemy II 平台提供的建模基础。第 12 章介绍软件体系结构，为读者进行 Ptolemy II 的扩展提供有效引导。第 13 章介绍在 Expression 角色中设定参数值和执行计算的表达式语言。第 17 章介绍由角色库提供的信号绘图功能角色。第 14 章介绍 Ptolemy II 类型系统。第 15 章介绍本体系统。第 16 章介绍网络接口，包括了将 Ptolemy II 模型输出到网站的机制和创建自定义 Web 服务器的机制。

软件体系结构

Christopher Brooks、Joseph Buck、Elaine Cheong、John S. Davis II、Patricia Derler、Thomas Huining Feng、Geroncio Galicia、Mudit Goel、Soonhoi Ha、Edward A. Lee、Jie Liu、Xiaojun Liu、David Messerschmitt、Lukito Muliadi、Stephen Neuendorffer、John Reekie、Bert Rodiers、Neil Smyth、Yuhong Xiong 和 Haiyang Zheng

本章介绍 Ptolemy II 的软件体系结构，方便读者进行自定义软件的扩展，如创建新的指示器和角色。更多细节可参考 Brooks et al. (2004) 的研究成果和 Ptolemy II 的源代码。本章假设读者熟悉 Java 语言，因为大部分 Ptolemy II 用此语言编写，并使用 UML 类图描述体系结构的关键特性。

12.1 包结构

Ptolemy II 是 Java 类组成的多个包的集合。包结构如图 12-1 所示。

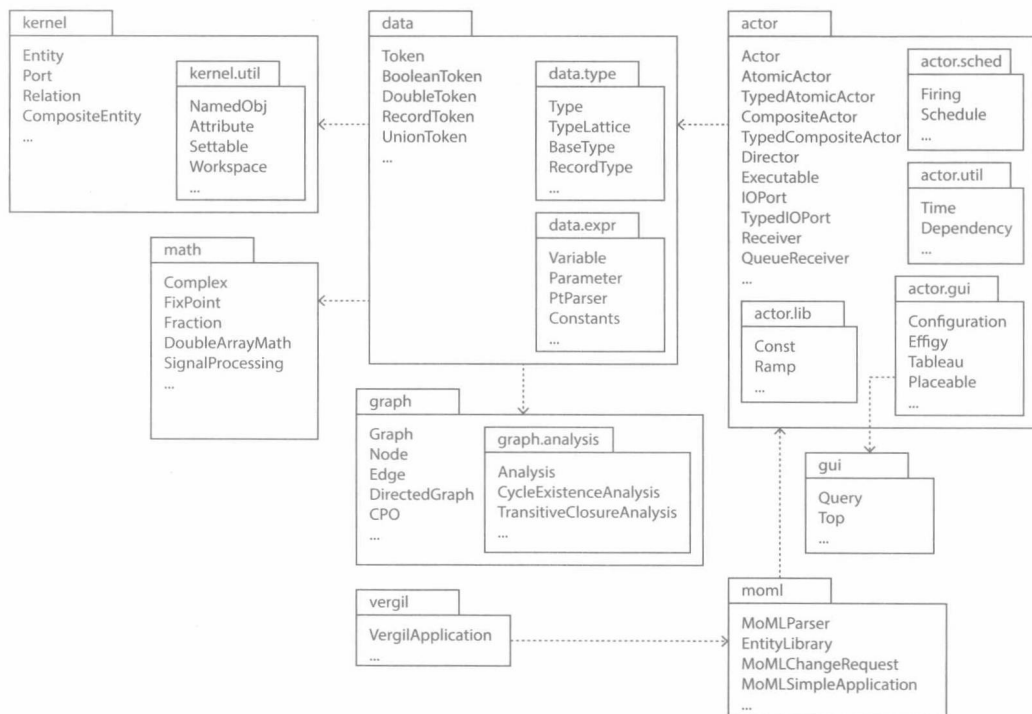


图 12-1 Ptolemy II 中的关键包及其包含的类

内核包 (Kernel): 内核包及其子包是 Ptolemy II 的核心。它定义了 Ptolemy 模型每个部分的基本类。内核包本身很小，其设计将在 12.2 节中介绍。该包定义了模型的结构，尤其指明层次关系，例如，组件和域的层次关系，以及一个模型的组件之间的内部联结。

数据包 (data): 数据包定义了将数据在模型的组件之间进行转移的类。Token 类尤为重要, 因为它是组件之间所有单元交换数据的基本类。Data.expr 包定义的表达式语言将在第 13 章详细介绍, 其使用表达式语言为模型中的参数赋值并建立参数之间的相互依赖关系。Data.type 包定义类型系统 (将在第 14 章介绍)。

数学包 (math): 数学包包含有一些对矩阵和向量进行运算的数学函数和方法, 还包括复数类、分数类, 以及定点数类。

图包 (graph): 该图包及其子包, graph.analysis, 提供了用于运算和分析数学图表的算法。该包为类型系统和其他模型分析工具提供了一些用于调度的核心算法。

角色包 (actor): 角色包 (详见 12.3 节) 包括角色的基本类和 I/O 端口, 其中角色被定义为通过 I/O 端口接收和发送数据的可执行实体。同时该包还包括为每个域定制的用来控制模型执行的基本类指示器 (Director)。角色包所包含的子包内容如下:

- actor.lib 包中包含一个庞大的角色库。
- actor.sched 包中包含为执行中的角色进行表示和调度构建的类。
- actor.util 包中包含核心 Time 类, 它实现了模型时间, 如 1.7.1 节所述。它还包含用以跟踪输出端口和输入端口之间依赖关系的类。
- actor.gui 包中包含管理用户界面的核心类, 包括可支持自定义结构的 Configuration (配置) 类和 Ptolemy II 的独立标记子集。Effigy 和 Tableau 类支持打开和查看模型和子模型。Placeable 接口和相关的类支持角色拥有自己的用户界面。

图形用户界面包 (gui): 该包提供用于模型组件的交互式编辑参数和窗口管理的用户界面组件。

Moml 包: 该包为 MOML 文件 (建模标记语言, Lee and Neuendorffer (2000) 提出, 格式为 XML, 用来存储 Ptolemy II 模型) 提供一个解析器。

Vergil 包: 提供 Vergil 的实现, 其中 Vergil 是 Ptolemy II 的图形用户界面 (详见第 2 章)。

还有许多其他的包和类, 这里涉及的内容概述了系统体系架构。下一节将讨论如何在内核包中定义模型的结构。

12.2 模型结构

计算机科学家在程序语言的语法和语义之间做了区分。程序语言语法特指构建有效程序的规则, 语义指的是程序的意义。同样的区分可以用到模型中。模型的语法是指它如何被表达, 而语义是指这一模型所代表的含义。

后来计算机科学家又在抽象语法和具体语法之间做了区分。模型的**抽象语法** (abstract syntax) 是指用何种结构来表示。例如, 模型可能以**图 (graph)**的形式表示, 图是结点和边的集合, 边连接结点。或者以**树 (tree)**的形式表示, 用来定义层级。Ptolemy II 的抽象语法广义上理解为一棵树 (表示模型的层级), 树的每一层是一个图 (用于规定组件之间的连接)。树结构保证了每一个结点都有一个确定的**容器 (container)**。

相反, **具体语法 (concrete syntax)** 是为了表示抽象语法而诞生的一种特殊表示方法。Vergil 的框图就是一种具体语法。MoML XML 标记语言是模型的文本语法 (通过字符串建立语法)。具体语法的所有结构的集合可以表示它的抽象语法。抽象语法限制了模型的结构, 具体语法以文本或者图的方式给出模型的描述。

抽象语法和具体语法都可以进行形式化定义。例如, 文本形式的具体语法可以由**巴科斯范式 (Backus-Naur Form, BNF)**给出, 计算机科学家对它很熟悉。实际上, BNF 是用于

这些端口进行交互。交互由关系调解。这些关系代表了通信路径。可以给所有这些对象（实体、端口、关系）分配属性。属性定义了它们的参数或者注解。端口和关系之间有链路，在元模型中被表示为关系类 Relation 和端口类 Port 之间的关联。

NameObj 包含一系列 Attribute（属性）的实例（也可能是空的）。Entity（实体）也包含 Port（端口）实例的集合（也可能是空的）。Port（端口）与 Relation（关系）实例相关联，关系定义了端口之间的连接。CompositeEntity 是一种包含了实体和关系实例的实体。图 12-4 显示了模型的层次结构。如前所述，角色（actor）是可执行实体，基于图 12-2 得出结论：AtomicActor 和 CompositeActor 实现执行（Executable）接口。指示器是 Director 类的执行实例，属性的子类。层次模型的每一层都有一个或者零个指示器。顶层总是有一个指示器。

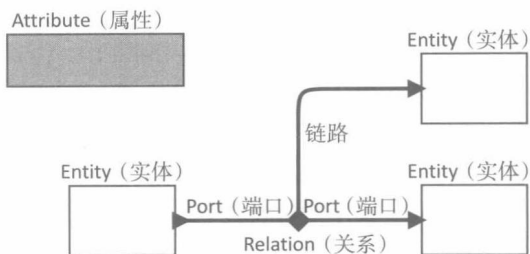


图 12-3 Ptolemy II 模型，标出了模型中对象的基元模型的类名称

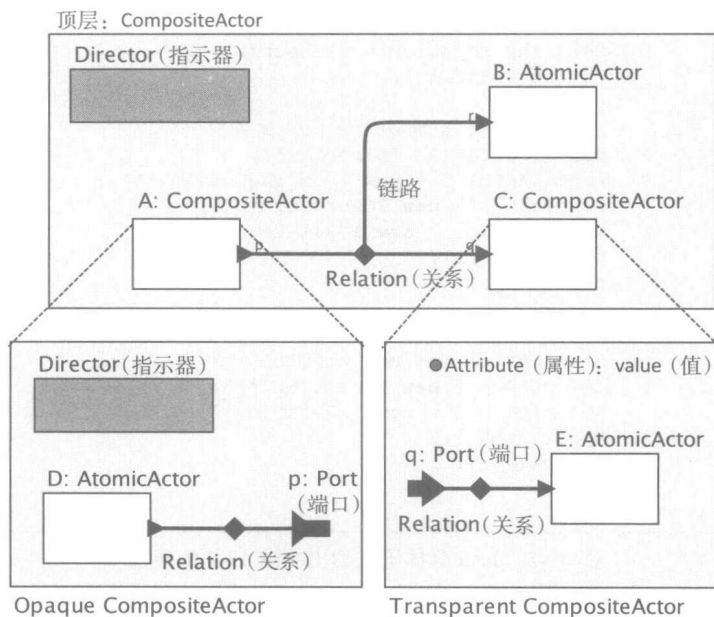


图 12-4 分层模型，标出了模型中对象的基元模型的类名称

例 12.1：图 12-4 是 Ptolemy II 层次模型的一个例子。该例用到了 Vergil 可视编辑器的具体可视化语法（visual syntax）。

图 12-4 中给出了 3 个不同的子模型和它们的层次关系。层次结构的顶层用“TopLevel : CompositeActor”标记，表示其名称是 TopLevel，并且它是 CompositeActor 的一个实例。TopLevel 包含一个指示器的实例，3 个角色和一个关系。角色 A 和 C 是复合角色，角色 B 是原子角色。3 个角色的端口与关系链接。复合角色的端口在图中出现了两次，一次在复合角色的外部，一次在内部。

图 12-4 中的框图使用了同一个模型的多种可能的具体语法中的一种。该模型也可以用 Java 语法定义，如图 12-5 所示；或者用 XML 标记语言（也称为 MoML）定义，如图 12-6 所示。这

三类语法都描述了模型的结构（这符合抽象语法）。下面我们将给出结构的某些意义，即语义。

```

1  import ptolemy.actor.AtomicActor;
2  import ptolemy.actor.CompositeActor;
3  import ptolemy.actor.Director;
4  import ptolemy.actor.IOPort;
5  import ptolemy.actor.IORelation;
6  import ptolemy.kernel.Relation;
7  import ptolemy.kernel.util.IllegalActionException;
8  import ptolemy.kernel.util.NameDuplicationException;
9
10 public class TopLevel extends CompositeActor {
11     public TopLevel()
12         throws IllegalActionException,
13             NameDuplicationException {
14         super();
15         // Construct top level.
16         new Director(this, "Director");
17         CompositeActor A = new CompositeActor(this, "A");
18         IOPort p = new IOPort(A, "p");
19         AtomicActor B = new AtomicActor(this, "B");
20         IOPort r = new IOPort(B, "r");
21         CompositeActor C = new CompositeActor(this, "C");
22         IOPort q = new IOPort(C, "q");
23         Relation relation = connect(p, q);
24         r.link(relation);
25
26         // Populate composite actor A.
27         new Director(A, "Director");
28         AtomicActor D = new AtomicActor(A, "D");
29         IOPort D_p = new IOPort(D, "p");
30         Relation D_r = new IORelation(A, "r");
31         D_p.link(D_r);
32         p.link(D_r);
33
34         // Populate composite actor C.
35         AtomicActor E = new AtomicActor(C, "E");
36         IOPort E_p = new IOPort(E, "p");
37         Relation E_r = new IORelation(C, "r");
38         E_p.link(E_r);
39         q.link(E_r);
40     }
41 }

```

图 12-5 Java 具体语法给出的图 12-4 的模型

```

1  <?xml version="1.0" standalone="no"?>
2  <!DOCTYPE entity PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
3      "http://ptolemy.eecs.berkeley.edu/xml/dtd/MoML_1.dtd">
4  <entity name="TopLevel" class="ptolemy.actor.CompositeActor">
5      <property name="Director" class="ptolemy.actor.Director"/>
6      <entity name="A" class="ptolemy.actor.CompositeActor">
7          <property name="Director" class="ptolemy.actor.Director"/>
8          <port name="p" class="ptolemy.actor.IOPort"/>
9          <entity name="D" class="ptolemy.actor.AtomicActor">
10             <port name="p" class="ptolemy.actor.IOPort"/>
11          </entity>
12          <relation name="r" class="ptolemy.actor.IORelation"/>
13          <link port="p" relation="r"/>
14          <link port="D.p" relation="r"/>
15      </entity>
16      <entity name="B" class="ptolemy.actor.AtomicActor">

```

图 12-6 MoML 具体语法给出的图 12-4 的模型

```

17      <port name="r" class="ptolemy.actor.IOPort"/>
18    </entity>
19    <entity name="C" class="ptolemy.actor.CompositeActor">
20      <property name="Attribute"
21        class="ptolemy.kernel.util.Attribute"/>
22      <port name="q" class="ptolemy.actor.IOPort"/>
23      <entity name="E" class="ptolemy.actor.AtomicActor">
24        <port name="p" class="ptolemy.actor.IOPort"/>
25      </entity>
26      <relation name="r" class="ptolemy.actor.IORelation"/>
27      <link port="q" relation="r"/>
28      <link port="E.p" relation="r"/>
29    </entity>
30    <relation name="r" class="ptolemy.actor.IORelation"/>
31    <link port="A.p" relation="r"/>
32    <link port="B.r" relation="r"/>
33    <link port="C.q" relation="r"/>
34  </entity>

```

图 12-6 (续)

12.3 角色语义和计算模型

许多 Ptolemy II 模型是面向角色的模型，即基于一组相互连接的角色。在面向角色的模型中，角色并发执行，通过端口在角色之间数据的传输。这意味着“并发执行”和数据在角色之间的传输方式依赖于角色正在运行的计算模式（Model of Computation, MoC）。在 Ptolemy II 中，计算模型是由放置于该位置的指示器定义的。

角色本身也可以成为一个 Ptolemy II 模型，这样的角色称为复合角色。包含监视器的复合角色是不透明的（opaque）；否则，称为透明的（transparent）。不透明的复合角色与非复合角色（即原子）的行为类似，它的内部结构在使用它的模型中是不可见的，它是一个黑盒。相反，透明复合角色从外部是完全可见的，并且就其本身而言是不可执行的。不透明复合角色（黑盒）对层次异构模型很重要，因为它们允许不同的计算模型嵌套在单一模型中。

正如区分抽象语法和具体语法一样，也可以把抽象语义和具体语义区分开。例如，考虑角色之间的通信，抽象语义关注的是通信发生（即角色发送一个令牌（token）给另一个角色），然而具体语义关注通信是“如何”发生的（例如，它是会话通信，还是异步消息传输，还是定点等）。一个指示器实现一个具体语义。抽象语义控制跨越不同层次（level）的指示器之间的交互。

Ptolemy II 给出了一个特殊的抽象语义，即角色抽象语义。它是指示器的互操作性的中心，也代表了构建异构模型的能力。角色抽象语义定义了角色行为的 3 个不同方面：执行控制、通信和时间模型。下面分别讨论它们^①。

12.3.1 执行控制

模型的整体执行是由 Manager 类的实例控制的。图 12-7 是由不透明复合角色构成的分层模型的执行序列示例，每个复合角色有一个指示器。如图 12-2 元模型所示，指示器是实现可执行的（Executable）接口的属性（Attribute）。如图 12-4 所示，在 Vergil 中，用绿色矩形表示。把指示器嵌入复合角色可以使该复合角色成为可执行的，因为指示器实现了 Executable（可执行的）接口，原子角色也能实现这种接口。

① 本书简略介绍了角色抽象语义。更详细的框架可以查阅 Tripakis et al. (2013) 和 Lee and Sangiovanni-Vincentelli (1998)。

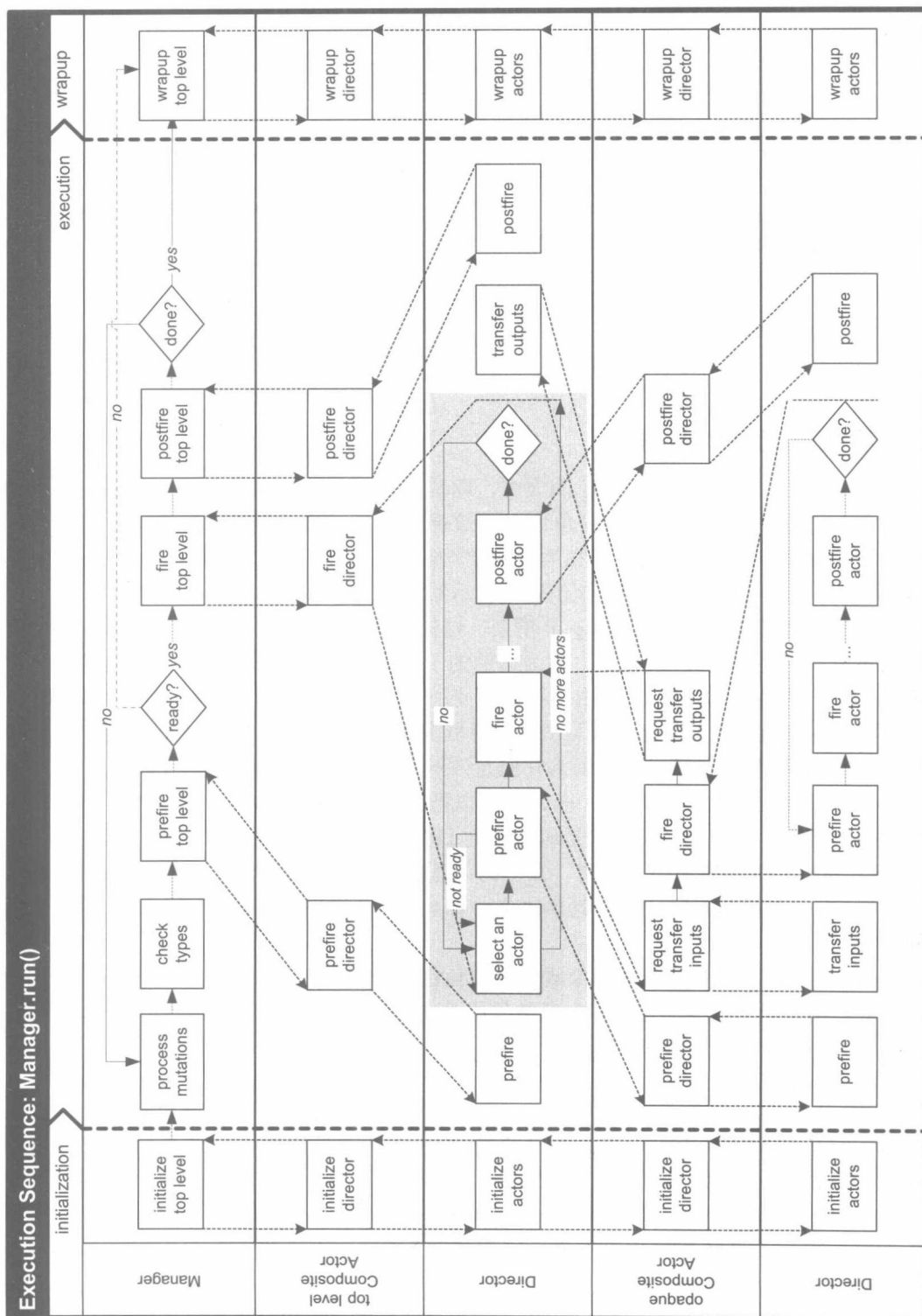


图 12-7 带有不透明复合角色的分层模型的执行

可执行 (Executable) 接口定义了实现计算角色抽象语义的行为。它们分为 3 个阶段：建立、迭代、打包。每个阶段进一步划分为多个子阶段。下面将介绍这些子阶段。

建立阶段分为预初始化和初始化行为，这些行为由可执行 (Executable) 接口实现。在预初始化行为中，角色的执行可能会影响静态分析（包括调度、类型推理和校验、代码生成等）的任何操作。复合角色可能会通过创建内部角色，改变自身的内部结构，例如在预初始化行为中。初始化阶段的初始化行为是进行参数初始化、重置本地状态、输出所有的初始化信息。在模型执行的过程中，角色的预初始化行为只进行一次，但是初始化行为可能进行多次。例如，如果语义需要重新初始化角色，那么初始化行为就可能会再次执行（如混合系统形式化 (Lee 和 Zheng, 2005)）。

迭代阶段是模型的最初执行阶段。在这个阶段，每个角色执行一个迭代序列，这些序列指（通常是有限的）能够使角色达到静止状态的计算。在复合角色中，计算模型决定一个角色的迭代对其他角色迭代行为的影响（比如它们是并发的还是交替的、如何调度等）。

为了协调各角色之间的迭代，一个迭代可分为预点火 (prefire)、点火 (fire) 和后点火 (postfire)。预点火（视情况）检测进行点火 (fire) 行为所需要的先决条件，如是否存在足够的输入。角色的主要计算是在点火行为中执行的，在这一行为中，读入输入数据、执行计算、产生输出数据。在执行过程中，一个角色可能会衍生出后续状态。后点火就是更新这种状态，以处理任意输入。角色抽象语义的一个重要组成部分是：只能在后点火中进行角色状态的更新，详见第 12 章补充阅读：为什么分为预点火、点火和后点火。

打包阶段是执行的最后阶段。即使执行因前面阶段出现意外而失败，总结也一定会执行。

补充阅读：为什么分为预点火、点火和后点火

虽然把角色执行的迭代阶段分为预点火、点火，后点火 3 个阶段并不明显，但对某些 Ptolemy II 模型却是很有必要的。同角色抽象语义定义的一致，点火行为读入输入数据，产生输出数据，且不改变角色的状态，状态的改变只有在后点火阶段出现。对具有固定点语义的 MoC，这种方法很有必要，其中 MoC 包括同步响应 (SR) 域和 Continuous (连续) 域。这样域的指示器通过重复点火角色直到一个定点到达才计算角色的输出。为保证确定性，在点火过程中保证每个角色的状态稳定。角色的状态只有在定点到达后才能更新。此时角色的所有输入都是已知的。直到后点火阶段这才出现。

然而，Ptolemy II 不是严格要求每个角色都遵守该协议。Goderis et al. (2009) 把面向角色的 MoC 分为抽象语义的 3 个类别：严格 (strict)、宽松 (loose) 和最宽松 (loosest)。在严格角色语义中，预点火、点火、后点火都是有限计算的，只有后点火 (postfire) 才能改变状态。在宽松角色语义中，状态的改变发生在点火的子阶段。在最宽松角色语义中，点火子阶段有可能不是有限的，其计算不会终止。

从某种意义上，符合严格角色语义的角色是最灵活的角色类型。该类型可以用在任何域中，包括 SR 域和连续 (Continuous) 域。这样的角色称作多态域 (domain polymorphic)。库中大部分角色都是多态域。只有符合宽松角色语义的角色可以和少数指示器（如数据流）一起使用。这些角色（列在域分类表中）。只有符合最宽松角色语义的角色仍然可以和少数指示器（如过程网络）一起使用。因此可以定义只能和一种指示器类型工作的角色。

与角色类似，指示器也实现相同的执行阶段。因此，在一个复合角色中嵌入指示器能够赋予该角色一个执行语义。如果指示器符合严格角色语义，那么这一复合角色就是域多态。这样的指示器支持 Ptolemy II 中层次异构的最灵活方式，因为在单个模型中，有不同 MoC 的多重指示器可能会分层结合。

12.3.2 通信

与角色的执行控制类似，角色的通信能力也是抽象语义的一部分。如前文所述，角色通过端口进行通信。对端口而言，可能是单端口，也可能是多重端口。如图 12-2 所示，每个角色都包含 IOPort 实例的端口，IOPort 是 Port 的子类。子类指定端口是用于输入还是输出。IOPort 子类提供两个主要方法：get 和 send。作为点火行为的一部分，角色用 get 方法检索输入，并执行计算；用 send 方法将结果发送到输出端口。对于多重端口，get 和 send 方法的整型参数指定一个通道。但是接收和发送究竟意味着什么？通信是否可以解释为 FIFO 队列、会话通信或者其他通信类型？这取决于指示器，而不是角色。

指示器通过创建一个接收方（receiver）并将其放在输入端口来决定角色之间如何通信，其中每个通信通道对应一个接收方。接收方是实现 Receiver（接收方）接口的对象，如图 12-8 所示。接口包括 put 方法和 get 方法。如图 12-9 所示，当某一角色调用输出端口的 send 方法时，输出端口将请求调用指定输入端口接收机的 put 方法。类似地，当角色调用输入端口的 get 方法时，输入端口将请求调用接收机的 get 方法。因此，由于指示器提供了接收方，所以指示可以进行发送和接收数据的控制。

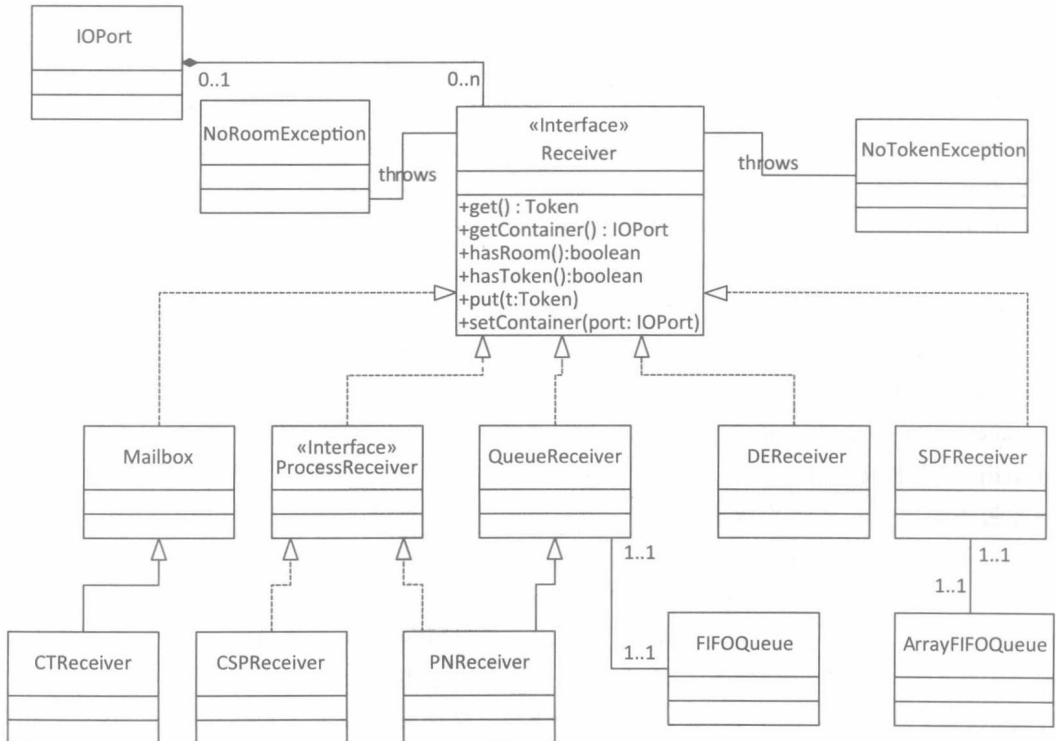


图 12-8 Ptolemy II 中的通信元模型

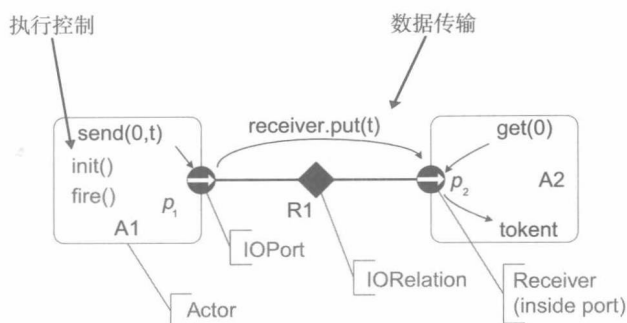


图 12-9 Ptolemy II 中的通信机制

接收器能实现 FIFO 队列、邮箱、全局队列会话。所有这些都与图 12-8 所示的元模型一致。指示器提供了能实现一种适合计算模型的通信机制的接收器。图 12-8 给出了一些接收器类。

例 12.2 PNReceiver 是 QueueReceiver 的一个子类，用于进程网络 (PN) 指示器。PNReceiver 的 put 方法将一个数据令牌 t 附加到 FIFO 队列并返回。当其返回之前，不能保证该信息已经被接收。PN 域实现非阻塞式写。它不用等待接收机做好接收准备的情况下就分发令牌，因此不会阻断模型的连续执行。

在 PN 中，每个角色运行自己的 Java 线程。角色接收信息后在不同的线程上异步运行，而不是发送令牌接收角色调用输入端口的 get 方法，该端口则调用 PNReceiver 的 get 方法。后者将阻断调用线程，直到 FIFO 队列至少接收到一个令牌。然后，它将返回队列的第一个令牌。因此，PNReceiver 实现了 PN 域所需要的阻塞式读入。阻塞式读入保证了 PN 模型是可确定性的。

12.3.3 时间

抽象角色语义的最后部分是时间概念。即角色用于计时域中时间的推进方式。

当角色点火时，可以从它的指示器获取当前时间，代码实现如下所示：

```
Time currentTime = getDirector().getModelTime();
```

由指示器返回的时间称为当前模型时间 (current model time)。如果角色只在后点火子阶段需要当前模型时间，那么就保证了返回的时间值非递减的。然而，如果在点火子阶段需要当前模型时间，就不能得到与后点火阶段一样的时间值。因为有些指示器 (如连续指示器，见第 9 章) 当收敛于定点时将推测进行时间的推进。在这样的例子中，当前模型时间会比 fire 的优先调用时间的值小。

角色可以通过调用其指示器的 fireAt 方法，在之后某个时间中被点火。指示器的责任是保证角色在规定的时间被点火。如果不能，指示器返回另一时间，角色可在这个时间被点火。然而角色不能假定它“下”次被点火的时间，所以有任意数量的中间点火。更有甚者，如果模型在点火规定的时间前执行完毕，那么角色在规定时间将不会被点火。

模型的层级对时间的管理是十分重要的。需要强调的是只有顶层指示器推进时间。模型中的其他指示器从它们上层指示器中获得当前模型时间。如果顶层指示器没有实现计时的计算模型，则时间不会被推进。

也许相反，即使不计时的域也提供对时间的访问。在层次模型中，除非不计时的域位于顶层，否则它将与层次结构上时间相关的操作授权给它的容器。

计时和不计时计算模型在层次结构中交错出现（见 1.7.1 节）。然而，有些特定组合不易理解。例如，如果顶层指示器不能推进时间，角色请求在未来某个时间点火，那么该请求将不能被满足。指示器的 `fireAt` 方法将返回一个时间，在该时间点火角色，这个时间将是时间 0，因为它从不会被推进。接受这一事实还是抛出一个异常来表示它与上层指示器不兼容，取决于角色自身。

如 1.7.1 节所述，模型中的时间可以非统一的推进。尤其在模态模型中（见第 8 章），时间的推进可以随时挂起（Lee and Tripakis, 2010）。在子模型中，本地时间与上层时间之间有单调非减的差值。该机制用来对子模型的时挂间起建模，如 8.5 节所述。

12.4 在 Java 中设计角色^①

Ptolemy II 中角色的功能有多种定义方式。最基础的机制是层次结构的使用，在这一机制中，一个角色定义为其他角色的复合角色。然而，对于实现复杂数学功能的角色，通常使用 Expression 角色更方便。它的功能用第 13 章描述的表达式语言来定义，也可以用 MatlabExpression 角色创建，通过将行为定义为 MATLAB 脚本的方式实现。在 Python 语言中，可以用 PythonActor 或者 PythonScript 进行角色定义，或者使用 Cal 角色定义语言来进行角色定义（Eker and Janneck, 2003）。但是，最灵活的方法是在 Java 中定义角色，这是本节讨论的重点。

如前所述，有些角色是多态域的，这意味着它们可以在多个域中进行操作。这里，本书关注为多态域创建的角色，也关注多态角色的创建，这些多态角色可以在多种类型的令牌数据上进行操作。当创建角色库时，域和数据多态性有助于最大化角色的可重用性，最小化重复代码。

代码重复也可以通过使用面向对象继承进行规避，继承也有助于增强角色之间的一致性。在默认库中的大多数角色扩展成基本类的公共集，使常用的端口和参数有了统一的名称。用公共基本类避免了不必要的差异，如将输入端口命名为“in”或者“inputSignal”或者“input”。这使得角色易于使用，它们的编码也更容易维护。为此，本书建议使用一个合理的深度类分层结构来提高一致性。划分子类和覆写已存在的角色比复制一个类并修改这个类要好很多。

注意，已存在的 Ptolemy II 角色的 Java 源代码，可以通过 Open Actor 上的菜单项找到，可以为新角色的定义提供有用的参考^②。

每一个角色都由 Java 写的源代码文件组成。图 12-10 给出了一个简单角色的源代码。该文本可以放在 Java 文件中进行编译，在 Vergil 中实例化，然后作为一个角色使用。创建一个新角色并在 Vergil 中使用的步骤：选择一个 Java 开发环境（例如，Eclipse），创建一个 Java 文件，将文件保存在 classpath^③中，然后在 Vergil 中实例化这个角色。实例化可以通过在菜单项 [Graph → Instantiate Entity] 打开的对话框中指定完整的类名来完成。例如，可

① 本节假定读者已掌握了一定的 Java 或面向对象编程知识。

② 如果你计划为 Ptolemy II 中的开源角色集提供定制的角色，那么请你务必遵循 Brooks and Lee (2003) 提出的编码风格。

③ classpath 由称为 CLASSPATH 的环境变量定义，在 Java 中用它搜索类定义。默认情况下，当运行 Vergil 时，如果有一个名为“ptolemyII”的目录在主目录中，那么该目录将在 classpath 中。可以把 Java 类文件存放在此目录下，Vergil 可以找到它们。

以把图 12-10 的文本复制到名为 Count.Java 的文件中，保存这个文件到 Ptolemy 安装的主目录下，然后在 Vergil 中使用 [Graph → Instantiate Entity] 创建该角色的实例。

```

1  import ptolemy.actor.TypedAtomicActor;
2  import ptolemy.actor.TypedIOPort;
3  import ptolemy.data.IntToken;
4  import ptolemy.data.expr.Parameter;
5  import ptolemy.data.type.BaseType;
6  import ptolemy.kernel.CompositeEntity;
7  import ptolemy.kernel.util.IllegalActionException;
8  import ptolemy.kernel.util.NameDuplicationException;
9
10 public class Count extends TypedAtomicActor {
11     /** Constructor */
12     public Count(CompositeEntity container, String name)
13         throws NameDuplicationException,
14             IllegalActionException {
15         super(container, name);
16         trigger = new TypedIOPort(this, "trigger", true, false);
17         initial = new Parameter(this, "initial", new IntToken(0));
18         initial.setTypeEquals(BaseType.INT);
19         output = new TypedIOPort(this, "output", false, true);
20         output.setTypeEquals(BaseType.INT);
21     }
22     /** Ports and parameters. */
23     public TypedIOPort trigger, output;
24     public Parameter initial;
25
26     /** Action methods. */
27     public void initialize() throws IllegalActionException {
28         super.initialize();
29         _count = ((IntToken)initial.getToken()).intValue();
30     }
31     public void fire() throws IllegalActionException {
32         super.fire();
33         if (trigger.getWidth() > 0 && trigger.hasToken(0)) {
34             trigger.get(0);
35         }
36         output.send(0, new IntToken(_count + 1));
37     }
38     public boolean postfire() throws IllegalActionException {
39         _count += 1;
40         return super.postfire();
41     }
42     private int _count = 0; /** Local variable. */
43 }

```

图 12-10 一个简单 Count 角色

在图 12-10 的源代码中，第 1 ~ 8 行指定角色依赖的 Ptolemy 类。可以查看这些类的源代码，在 Java 开发环境（如 Eclipse）中可以很容易地查看这些文件。

第 10 行进行 Count 类的定义，这个类是 TypedAtomicActor 的子类。（TypedAtomicActor 是大部分 Ptolemy II 角色的基本类，这些角色的端口和参数有确定的类型）。这个特殊的角色可以代替子类 Source 或 LimitedFiringSource，这两者都可以提供所需要的端口。但是这里，为了直观些，给出了端口的定义。对角色有特殊作用的基本类是 Transformer，如图 12-11 所示。对于有一个输入端口和一个输出端口的角色，这是合理的选择。

第 12 ~ 20 行给出了构造函数，构造函数是创建类实例化的 Java 过程。传递给构造函

数的参数定义角色名和角色放置的位置。在构造函数体中，第 15 行创建了一个名为 `trigger` 的输入端口。`TypedIOPort` 构造函数的第 3 和第 4 个参数指出这个端口是输入端口而不是输出端口。按惯例，在 `Ptolemy II` 中，作为公有域每个端口可以是可见的（第 22 行定义）。这个公有域的名称与第 15 行给出的构造函数的参数的名称是匹配的，要使面向角色的类（见 2.6 节）正常工作，需保证这些名称的匹配。

```

1  public class Transformer extends TypedAtomicActor {
2
3      /** Construct an actor with the given container and name.
4       *   @param container The container.
5       *   @param name The name of this actor.
6       *   @exception IllegalArgumentException If the actor
7       *   cannot be contained by the proposed container.
8       *   @exception NameDuplicationException If the container
9       *   already has an actor with this name.
10      */
11     public Transformer(CompositeEntity container, String name)
12         throws NameDuplicationException,
13             IllegalArgumentException {
14         super(container, name);
15         input = new TypedIOPort(this, "input", true, false);
16         output = new TypedIOPort(this, "output", false, true);
17     }
18
19     //////////////////////////////////////////
20     ///          ports and parameters          ///
21
22     /** The input port. This base class imposes no type
23      *   constraints except that the type of the input
24      *   cannot be greater than the type of the output.
25      */
26     public TypedIOPort input;
27
28     /** The output port. By default, the type of this output
29      *   is constrained to be at least that of the input.
30      */
31     public TypedIOPort output;
32 }

```

图 12-11 对于具有一个输入和一个输出的角色，`Transformer` 是一个有用的基类

第 16 行进行参数的定义。再次按惯例，参数与名称匹配的公共域，如第 23 行所示。第 17 行指定了参数的数据类型，限制了它的可能值。

第 18 和 19 行创建输出端口并设置其数据类型。在该角色中，没有条件限制 `trigger` 输入的类型，所以任何数据类型都是可接受的。

第 26 ~ 29 行给出的 `initialize` 方法把私有局部变量 `_count` 初始化为 `initial` 参数的值。按 `Ptolemy II` 的习惯，私有变量和保护变量的名称以下划线开始。这里转换为 `IntToken` 是安全的，因为参数的类型限制为整数。

第 30 ~ 36 行的 `fire` 方法读输入端口，如果这个端口被连接（也就是，如果它有大于零的宽度）并且有一个令牌。在某些域中，如 `DE`，读输入令牌是很重要的，即使这些令牌不会被用到。特别地，`DE` 指示器将重复点火一个角色，这个角色在它的输入端口有未消耗的令牌。读输入数据失败将导致无限的激活序列。第 35 行发送一个输出令牌。

第 37 ~ 40 行的 `postfire` 方法通过增加私有变量 `_count` 的数值更新该角色的状态。如

上所述，更新 `postfire` 中的状态而不是 `fire` 中的状态使带指示器（如 `SR`、`Continuous`）的角色可以被使用，因此可以重复点火一个角色直到达到一个定点。

12.4.1 端口

按惯例，端口是角色的公有成员。它们代表了输入和输出通道的集合，通过这个集合令牌能够传送到其他端口。图 12-10 表示了如何定义端口为公有域以及如何在构造函数中将其实例化。这里，介绍了一些对创建端口有用的选项。

1. 多重端口和单端口

一个端口可以是一个单端口或者一个多重端口，默认情况下，一个端口是一个单端口，可以用如下方式声明为多重端口：

```
portName.setMultiport(true);
```

每个端口有一个的宽度，它对应于它的通道的数目。如果一个端口没有被连接，那么宽度为 0。如果一个端口是一个单端口，那么宽度是 0 或者 1。如果一个端口是一个多重端口，那么宽度大于 1。

2. 读和写

用如下语法可以把数据（压缩于一个令牌中）发送到输出端口的特定通道：

```
portName.send(channelNumber, token);
```

`channelNumber` 从 0 开始，对应着第一个通道。端口的宽度（通道的数目）如下述获得：

```
int width = portName.getWidth();
```

如果端口没有连接，那么令牌不会被发送到任何地方。`send` 方法将简单地返回。注意，一般情况下，如果通道的数目引用一个不存在的通道，那么 `send` 方法简单地返回而不会抛出异常。相反，试图从不存在的输入通道读取令牌将导致异常。

令牌可以发送到端口的所有输出通道

```
portName.broadcast(token);
```

可以指定令牌值，然后通过以下语句发送它：

```
portName.send(channelNumber, new IntToken(integerValue));
```

令牌可以从通道读取：

```
Token token = portName.get(channelNumber);
```

可以从端口的通道 0 读取，并提取数据值（假设该类型是已知的）：

```
double variableName = ((DoubleToken)
    portName.get(0)).doubleValue();
```

可以查询输入端口来确定 `get` 是否成功（令牌是否有效）：

```
boolean tokenAvailable = portName.hasToken(channelNumber);
```

也可以查询输出端口来确定 `send` 是否成功：

```
boolean spaceAvailable = portName.hasRoom(channelNumber);
```

尽管有许多域（如 `SDF` 和 `PN`），但该选项都是有效的。

3. 端口之间的相关性

作为调度模型执行过程的一部分许多 Ptolemy II 域执行模型的拓扑分析。例如，SDF 构建确定角色调用顺序的静态调度。DE、SR 和连续 (Continuous) 都检测角色之间的数据依赖，以便优先响应同时事件。在所有这些情况下，指示器需要关于角色行为的附加信息以便执行分析。在本节中，解释如何提供该附加信息。

假设正在设计一个角色，它的输入端口不需要令牌，以便当它点火时，在它的输出端口产生一个令牌。获得这些信息对指示器是很有用的。这些信息可以表示在角色的 Java 代码中。例如，MicrostepDelay 角色声明输出端口与输入端口是相互独立的：

```
public void declareDelayDependency()
    throws IllegalArgumentException {
    _declareDelayDependency(input, output, 0.0);
}
```

默认情况下，假设每个输出端口依赖于所有的 input 端口。通过定义以上方法，MicrostepDelay 角色警告指示器这个默认是不可用的。在 input 与 output 端口之间有延迟。这里延迟声明为 0.0，称为微步延迟。调度程序可以使用该信息给角色的执行排序和解决因果循环。对于不使用依赖信息的域 (例如，SDF)，上述方法并不适用。因此，这些声明有助于最大化在多种域中角色的重用性。

4. 端口生产和消费率

有些域 (尤其是 SDF) 利用角色端口生产和消费率的信息。如果角色的创建者没有做特别的声明，SDF 指示器将假设一旦点火，角色在每个输入端口仅需要并消耗一个令牌，在每个输出端口产生一个令牌。为了消除这种假设，创建者需要在端口上包含一个名为 tokenConsumptionRate (输入端口) 或者 tokenProductionRate (输出端口) 的参数。这些参数的值都是整数，并且指定了点火过程中消耗或产生的令牌数目。这些参数的值可以通过表达式赋值，表达式依赖于角色的其他参数。与之前的例子类似，这些参数对不使用这些信息的域没有影响，但是对使用这些信息的域 (如 SDF) 中的角色产生影响。

SDF 中的反馈回路在回路中至少需要一个角色以便在 initialize 方法中产生令牌。为了警告角色包括这种能力的 SDF 调度程序，相关的输出端口必须包含一个整数值的参数 (命名为 tokenInitProduction)，该参数指定最初产生的令牌数目。SDF 调度程度将用该信息来确定一个循环模型是否发生死锁。

12.4.2 参数

与端口类似，按照惯例参数，也是角色的公有成员。公有成员的名称需要与传递给参数构造函数的名称相匹配，采用与端口相同的方式指定类型约束或参数。

角色通过调用 attributeChang 方法知道参数值发生了变化。若角色需要检查参数值是否有效，则可以通过重写 attributeChang 方法。考虑图 12-12 中例子的 PoissionClock 角色。这个角色根据泊松过程产生计时事件。其中一个参数是 meanTime，它表示事件之间的平均时间。与构造函数中的声明一样，参数必须是双精度的正数。如第 21 ~ 25 行所示，角色强制实施该约束，如果给出一个非正值，那么将抛出异常。

```
1 public class PoissionClock extends TimedSource {
2     public PoissionClock(CompositeEntity container, String name)
```

图 12-12 使用 attributeChanged 验证参数值

```

3         throws NameDuplicationException,
4             IllegalArgumentException {
5         super(container, name);
6         meanTime = new Parameter(this, "meanTime");
7         meanTime.setExpression("1.0");
8         meanTime.setTypeEquals(BaseType.DOUBLE);
9         ...
10    }
11    public Parameter meanTime;
12    public Parameter values;
13
14    /** If the argument is the meanTime parameter,
15     *  check that it is positive.
16     */
17    public void attributeChanged(Attribute attribute)
18        throws IllegalArgumentException {
19        if (attribute == meanTime) {
20            double mean = ((DoubleToken)meanTime.getToken())
21                .doubleValue();
22            if (mean <= 0.0) {
23                throw new IllegalArgumentException(this,
24                    "meanTime is required to be positive."
25                    + " Value given: " + mean);
26            }
27        } else if (attribute == values) {
28            ArrayToken val = (ArrayToken)
29                (values.getToken());
30            _length = val.length();
31        } else {
32            super.attributeChanged(attribute);
33        }
34    }
35    ...
36    }
37    ...
38 }

```

图 12-12 (续)

AttributeChanged 方法也可以用来缓存参数的当前值，第 26 ~ 28 行所示。

12.4.3 端口和参数耦合

通常，在角色设计过程中，很难为端口或参数指定一个量。幸运的是，存在两种选择来进行设计。Ramp 角色就是这样的例子，代码如图 12-13 所示。这个角色从参数 init 给出的初始值开始，然后这个值按步长增加 step。step 的值可以由名为 step 的参数或端口指定。如果端口是未连接的，那么则由 init 参数提供初始值。如果端口是连接的，那么由 init 参数提供初始默认值，然后这个值被到达端口的值取代。

```

1 public class Ramp extends SequenceSource {
2     public Ramp(CompositeEntity container, String name)
3         throws NameDuplicationException,
4             IllegalArgumentException {
5         super(container, name);
6         init = new Parameter(this, "init");
7         init.setExpression("0");
8         step = new PortParameter(this, "step");

```

图 12-13 Ramp 角色的代码片段

```

9         step.setExpression("1");
10        ...
11    }
12    public Parameter init;
13    public PortParameter step;
14    public void attributeChanged(Attribute attribute)
15        throws IllegalArgumentException {
16        if (attribute == init) {
17            _stateToken = init.getToken();
18        } else {
19            super.attributeChanged(attribute);
20        }
21    }
22    public void initialize() throws IllegalArgumentException {
23        super.initialize();
24        _stateToken = init.getToken();
25    }
26    public void fire() throws IllegalArgumentException {
27        super.fire();
28        output.send(0, _stateToken);
29    }
30    ...
31    public boolean postfire() throws IllegalArgumentException {
32        step.update();
33        _stateToken = _stateToken.add(step.getToken());
34        return super.postfire();
35    }
36    private Token _stateToken = null;
37 }

```

图 12-13 (续)

当参数值保存到 MoML 文件时，它与包含 Ramp 角色的模型存储在一起。相反，在模型执行过程中到达端口的任何值都不会被存储。因此，参数给出的默认值是固定的，然而到达端口的值是短暂的。

为了能够同时应用参数和端口，Ramp 角色在它的构造函数中创建了一个 PortParameter 类的实例，如图 12-13 所示。这个参数创建了一个同名的关联端口。postfire 方法首先在 step 上调用 update，然后把它的值添加到状态上。调用 update 对从关联输入端口读入有副作用，并且若这里有一个令牌，则更新参数的值。为了保证在关联输入端口的有效输入令牌都被消耗，在读 Portparameter 的值之前需要先调用 update。

12.5 小结

本章简要介绍了 Ptolemy II 的软件架构。介绍了构成 Ptolemy II 类的整体布局，以及内核包中的主要类是如何定义模型的结构。同时还解释了角色包中的主要类是如何定义模型的执行的。最后，简要介绍了如何在 Java 中写定制的角色。

表 达 式

Christopher Brooks、Thomas Huining Feng、Edward A. Lee、Xiaojun Liu、Stephen Neuendorffer、Neil Smyth 和 Yuhong Xiong

在 Ptolemy II 中，模型通过组合不同的角色来实现特定计算。然而，许多计算并不适合用这样的方式来指定。比如计算一个简单的代数表达式 $\sin[2\pi(x-1)]$ 。可以通过在框图中组合不同角色来表示这个计算，但用文本方式直接给出更为便捷。

Ptolemy II 表达式语言 (expression language) 提供了用文本方式指定代数表达式并计算结果的基础功能。表达式语言用于指定状态机中的参数值、条件与动作，以及 Expression 角色执行的计算。事实上，表达式语言是 Ptolemy II 中通用基础结构的一部分，程序员可以用其扩展 Ptolemy II 系统。本章将描述从用户而不是程序员的角度来如何使用表达式。

Vergil 提供了一个交互式表达式计算器 (expression evaluator)，可通过菜单命令 [File → New → Expression Evaluator] 来访问。如图 13-1 所示，该操作类似于一个交互式 shell 命令。它支持查看命令历史。输入向上箭头或 Control-P 可访问之前输入的表达式，输入向下箭头或 Control-N 可返回。表达式计算器可以检验表达式非常有用。

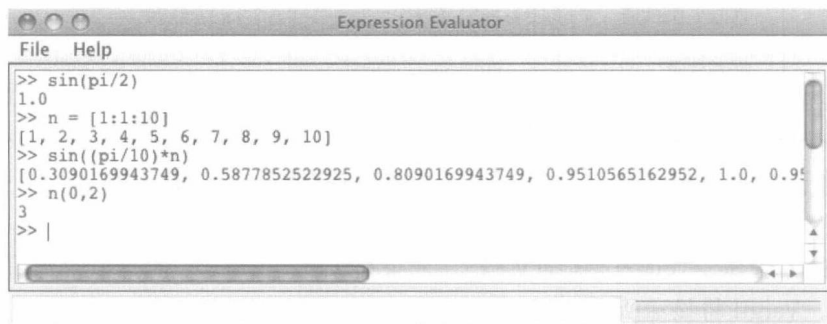


图 13-1 表达式计算器

13.1 简单算术表达式

13.1.1 常量与直接值

最简单的表达式是一个常量，它可以由常量的符号名称或一个直接值表示。默认情况下，支持的常量符号名称有：

PI、Pi、e、E、true、false、i、j、NaN、Infinity、PositiveInfinity、NegativeInfinity、MaxunsignedByte、MinunsignedByte、Maxshort、Minshort、Maxint、Minint、Maxlong、Minlong、Maxfloat、Minfloat、Maxdouble 以及 Mindouble。例如，

PI/2.0

是一个有效的表达式，它包括符号名称“PI”和直接值“2.0”。常量 i 和 j 是值为 $\sqrt{-1}$ 的虚数 (imaginary number)。常量 NaN 表示“无意义的数字”，例如除法 $0.0/0.0$ 的结果。常量 Infinity 表示除法 $1.0/0.0$ 的结果。以“Max”和“Min”开头的常量是对应类型的最大值和最小值。

不带小数点的数值 (如“10”或“-3”) 是整数 (int 类型)。带小数点 (如“10.0”或“3.14159”) 的数值是双精度 (double) 类型。后面有“f”或“F”的数值是浮点型 (float) 类型。不带小数点且后缀为“l”或“L”的数值是长整 (long) 型。不带小数点且后缀为“s”或“S”的数值是短整 (short) 型。无符号整数后跟“ub”或“UB”是无符号字节 (unsignedByte) 型，如“5ub”。unsignedByte 类型的值在 0 ~ 255 之间，注意，这与 Java 字节不相同，其值在 -128 ~ 127 之间。int、long、short 或 unsignedByte 类型的数值，可以表示为十进制、八进制或十六进制。数字以“0”开头的是八进制数。数字以“0x”开头的是十六进制数字。例如，“012”和“0xA”都是整数 10。

Complex 型是通过给 double 型附加一个“i”或“j”作为虚部来定义的。这给出了一个纯虚 Complx 型，然后它可以利用 Token 类的多态操作来创建一个普通复数。因此，复数可以写作 $2+3i$ 或者 $2+3*i$ 。

此外，还支持面值字符串常量。双引号之间的任何值“...”，解释为一个字符串常量。下面是内置的字符串值常量的定义：

变量名	说明	JVM 属性名	实例
PTII	Ptolemy II 安装目录	ptolemy.ptII.dir	c:\tmp
HOME	用户主目录	user.home	c:\Documents and Settings\you
CWD	当前工作目录	user.dir	c:\ptII
TMPDIR	临时目录	java.io.tmpdir	c:\Documents and Settings\you\Local Settings\Temp\
USERNAME	用户账号名	user.name	ptolemy

这些变量的值是由相应的 Java 虚拟机 (JVM) 属性给出的，如 HOME 的 user.home。属性 user.dir 和 user.home 是 Java 中的标准。它们的值与平台相关，详细内容请参阅 java.lang.System 类中的方法 getProperties 的文档^①。如果在 (图表编辑器) 的菜单中调用 [View → JVM Properties]，Vergil 将显示所有的 Java 属性。

当 Vergil 或任何其他 Ptolemy II 可执行文件启动时，ptolemy.ptII.dir 属性将会自动进行设置。当通用以下语法使用 java 命令启动 Ptolemy II 过程时，还可以设置它：

```
java -Dptolemy.ptII.dir=${PTII} classname
```

其中 classname 是 Java 应用程序的完整类名。同样也可以设置表中的其他变量。例如，为了在一个特定目录中调用 Vergil，使用以下命令：

```
java -cp /ptII -Duser.dir=/Users/eal \
    ptolemy.vergil.VergilApplication
```

① 需要注意的是，user.dir 和 user.home 通常在没有符号的小程序中无法读取，在这种情况下，试图在表达式中使用这些变量将导致异常。

表 13-1 constants 函数的返回值

变量名	值	变量名	值
CLASSPATH	" xxxxxxCLASSPATHxxxxxx "	CWD	" /Users/eal "
E	2.718281828459	HOME	" /Users/eal "
Infinity	Infinity	MaxDouble	1.797693134862316E308
MaxFloat	3.402823466385289E38	MaxInt	2147483647
MaxLong	9223372036854775807L	MaxShort	32767s
MaxUnsignedByte	255ub	MinDouble	4.9E-324
MinFloat	1.401298464324817E-45	MinInt	-2147483648
MinLong	-9223372036854775808L	MinShort	-32768s
MinUnsignedByte	0ub	NaN	NaN
NegativeInfinity	-Infinity	PI	3.1415926535898
PTII	" /ptII "	PositiveInfinity	Infinity
TMPDIR	" /tmp "	USERNAME	" eal "
backgroundColor	{0.9, 0.9, 0.9, 1.0}	boolean	false
complex	0.0 + 0.0i	double	0
e	2.718281828459	fALSE	false
fixedpoint	fix(0, 2, 2)	float	0.0f
general	present	i	0.0 + 1.0i
int	0	j	0.0 + 1.0i
long	0L	matrix	[]
nil	nil	null	object(null)
object	object(null)	pi	3.1415926535898
scalar	present	short	0s
string	""	true	true
unknown	present	unsignedByte	0ub
xmltoken	null		

-cp 选项指定类路径 (classpath) (必须包括 Ptolemy II 的根目录), -D 选项指定设置的属性, 最后一个参数是包含调用 Vergil 的 main 方法的类。

constants 通用函数返回带有所有全局定义常量的记录。如果打开表达式计算器并调用这个函数, 将看到它的值与图 13-1 中的一样。

13.1.2 变量

表达式可以用于表达式作用域内的变量标识符。例如,

```
PI*x/2.0
```

若 “x” 是作用域内的变量。在表达式计算器中, 作用域内的变量包括内置的常量和前面已经进行的赋值。例如,

```
>> x = pi/2
1.5707963267949
>> sin(x)
1.0
```

在 Ptolemy II 模型中, 作用域中的变量包括在层次结构的同层或高层定义的所有参数。例如, 如果一个角色的参数 “x” 的值为 1.0, 那么相同角色的另一参数有值为 “PI*

$x/2.0$ ”的表达式，其值为 $\pi/2$ 。

如果角色 X 中的参数 P ，其中 X 包含在复合角色 Y 中。 P 的表达式的作用域包括 X 和 Y 中包含的所有参数，以及 Y 的容器和 Y 包含的其他角色。也就是说，作用域包括定义在层次结构中的任何参数。

可以通过右击角色来为角色（复合的或者单一的）添加参数，选择 [Customize → Configure]，然后单击 “Add”，或从 Utilities 库中拖出参数。因此，可以在任何作用域内添加变量，就像任何函数编程语言中使用 “let” 结构的作用一样。

有时候，访问不在作用域中的参数也是可行的。表达式语言支持一种有限语法，该语法允许访问某些超出作用域的变量。如果在表达式中将名为 x 的变量写成 $A::x$ ，而不是寻找 x 的作用域，那么解释器会在作用域内查找名称为 A 的容器并在 A 中查找名为 x 的参数。从当前容器或它的任何容器向下到达层次结构的一个层是允许的。

13.1.3 运算符

算术运算符包括 $+$ 、 $-$ 、 $*$ 、 $/$ 、 $^$ 和 $\%$ 。大多数算术运算符可以对多种数据类型进行运算，包括数组、记录（records）以及矩阵。 $^$ 运算符计算“乘方”或求幂运算，其中指数只能是可以无损地转化成整型（如 int、short 或 unsignedByte）的类型。

unsignedByte、short、int 和 long 类型只能表示整数。这些类型的运算是整数运算，有时会产生意想不到的结果。例如， $1/2$ 得到 0，因为 1 和 2 是整数，而 $1.0/2.0$ 得到 0.5。当使用求幂运算符 “ $^$ ” 含有负值数时，同样可以产生意想不到的结果。例如， 2^{-1} 得到 0，因为结果是计算 $1/(2^1)$ 得到的。

$\%$ 运算符是模或求余运算。结果是相除后的余数。结果的符号与被除数（左边的参数）相同。例如，

```
>> 3.0 % 2.0
1.0
>> -3.0 % 2.0
-1.0
>> -3.0 % -2.0
-1.0
>> 3.0 % -2.0
1.0
```

结果的量级总是小于除数（右边参数）的量级。注意，当这个运算符用于双精度数时，其结果与 remainder 函数产生的结果不相同（见表 13-6）。例如，

```
>> remainder(-3.0, 2.0)
1.0
```

remainder 函数执行 IEEE 754 标准求余运算。它采用了舍入除法，而不是截断除法，因此符号可正可负，取决于复杂的运算规则（见 13.4.8 节）。例如，

```
>> remainder(3.0, 2.0)
-1.0
```

当操作数涉及两个不同类型时，表达式语言需要决定使用哪个类型来完成运算。如果某一类型可以无损地转换成另一种，那么就将此类型转换为另一种类型。例如，int 可以无损地转换成 double，所以 $1.0/2$ 首先将 2 转换为 2.0，得到结果 0.5。在标量中，unsignedByte 可以转换为其他的类型，short 类型可以转换成 int，int 可以转换成 double，float 可以转换

成 `double`, `double` 可以转换为 `complex`。注意, `long` 类型不能无损地转换为 `double`, 反之亦然, 所以像 `2.0/2L` 这样的表达式产生以下错误信息:

```
Error evaluating expression "2.0/2L"
in .Expression.evaluator
Because:
divide method not supported between ptolomy.data.DoubleToken
'2.0' and ptolomy.data.LongToken '2L' because the types are
incomparable.
```

类似地, `long` 不能转换为 `double`, `int` 也不能转换为 `float`, 反之亦然。

所有标量类型都限制了精度和大小。因此, 算术运算面临下溢 (`underflow`) 和上溢 (`overflow`) 的问题。

- 对于双精度浮点数, 上溢会导致相应的正无穷大或负无穷大, 下溢 (即精度并不足以代表结果) 会产生空值。
- 对于整数和定点类型, 上溢导致回绕。例如, `Maxint` 的值是 2147483647, 但是表达式 `Maxint+1` 产生 -2147483648。同样, 当 `MaxunsignedByte` 的值为 255 时, `MaxunsignedByte+1` 的值为 0。但是, 注意, `MaxunsignedByte+1` 产生 256, 是 `int` 类型, 而不是 `unsignedByte`。这是因为 `MaxunsignedByte` 可以无损地转换为 `int`, 所以这个加法是 `int` 类型的加法, 而不是 `unsignedByte` 类型的加法。

按位运算符是 `&`、`|`、`#` 和 `~`。它们可对 `boolean`, `unsignedByte`、`short`、`int` 和 `long` (但不能对 `fixedpoint` (定点类型)、`float`、`double` 或 `complex` (复数)) 进行运算。运算符 `&` 是按位与运算, `~` 是按位取反, `|` 是按位或, `#` 是按位异或。

关系运算符是 `<`、`<=`、`>`、`>=` 以及 `!=`。它们返回类型为 `boolean` 型。可以的话一般这些关系运算符只检验这些值的关系, 而忽略其类型。例如,

```
1 == 1.0
```

返回 `true`。如果希望同时检查类型是否相同, 数值是否相等, 则调用 `equals` 方法, 例如

```
>> 1.equals(1.0)
false
```

布尔值表达式可以用来设置条件值。该语法是

```
boolean ? value1 : value2
```

若该布尔表达式为 `true`, 则表达式的值为 `value1`, 否则, 为 `value2`。逻辑布尔运算符包括 `&&`、`||`、`!`、`&` 和 `|`。它们的运算数和返回值都是 `boolean` 类型。逻辑 `&&` 和逻辑 `&` 之间的区别在于, `&` 对所有的运算数进行计算而忽略其值是否与之相关。类似地, 逻辑 `||` 和 `|` 也是这样。这种方法借鉴了 `Java`。因此, 例如, 表达式 `false && x` 的结果是 `false` 而忽略 `x` 是否定义。另一方面, 如果 `x` 未定义则 `false & x` 将抛出异常。

`<<` 和 `>>` 运算符分别执行算术左移和算术右移。`>>>` 运算符执行无符号逻辑右移。它们的运算数类型是 `unsignedByte`、`short`、`int` 以及 `long`。

13.1.4 注释

在表达式语言中, `/*...*/` 中的内容可以忽略, 因此可在其中插入注释。

13.2 表达式的应用

Ptolemy II 中的表达式用于给参数赋值, 以便指定 `Expression` 角色实现了输入/输出函

数，并指定状态机的条件和动作。

13.2.1 参数

角色的大部分参数可以同表达式一样赋值^①。表达式中的变量引用作用域中内的其他参数，这些参数包含于同一个容器或层次结构中的容器中。也可以引用作用域扩展属性中那些包含变量定义单元的变量，这一点将在 13.7 节介绍。如第 2 章所述，给角色增加参数是比较直观的。

13.2.2 端口参数

可以定义一个具有端口功能的参数。这样的 PortParameter（端口参数）提供了一个默认值，类似于指定其他参数的值。但是当相应的端口接收数据，默认值就会被端口提供的值覆盖。因此，这个目标（object）函数就像参数和端口。访问 PortParameter 的当前值与访问任何其他参数一样。它的当前值将是默认值或端口最近接收的值。

PortParameter 可以包含在原子角色或复合角色中。把它从 utilities 库拖到模型中，可将 PortParameter 添加到复合角色中，如图 13-2 所示。

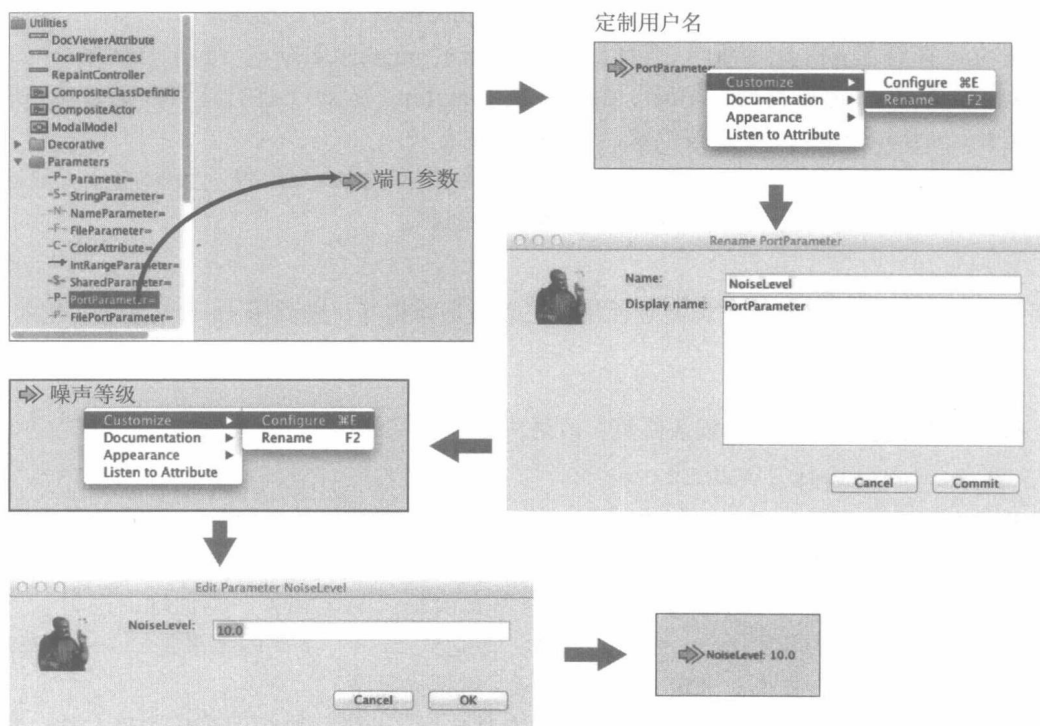


图 13-2 既是端口又是参数的 PortParameter。为了在复合角色中使用它，将它拖进角色，将它的名称改为有意义的名称并设置默认值

为了使用方便，必须给 PortParameter 提供一个名称（默认的名称“PortParameter”非

^① 其中有一部分参数例外，那就是严格的字符串参数。这类参数的值为逐个的字符串，而不是可以通过表达式描述的参数。例如那些可以通过表达式描述的函数参数，只需要利用三角函数：“sin”，“cos”，“tan”，“asin”，“acos”和“atan”就可以很好的对其进行赋值。

常不引人注目)。右击图标并选择 [Customize→Rename] 可对其改名, 如图 13-2 所示。在图中, 将名称设置为 “noise-Level”, 然后通过双击设置默认值。图中, 默认值设置为 10.0。

库角色使用 PortParameter 的一个例子是 Sinewave 角色, 它可以在 Vergil 的 Sources→SequenceSources 库中找到, 如图 13-3 所示。若双击该角色, 可以给频率 (frequency) 和相位 (phase) 设置默认值。也可以通过相应端口来设置 (见下面灰色填充的)。

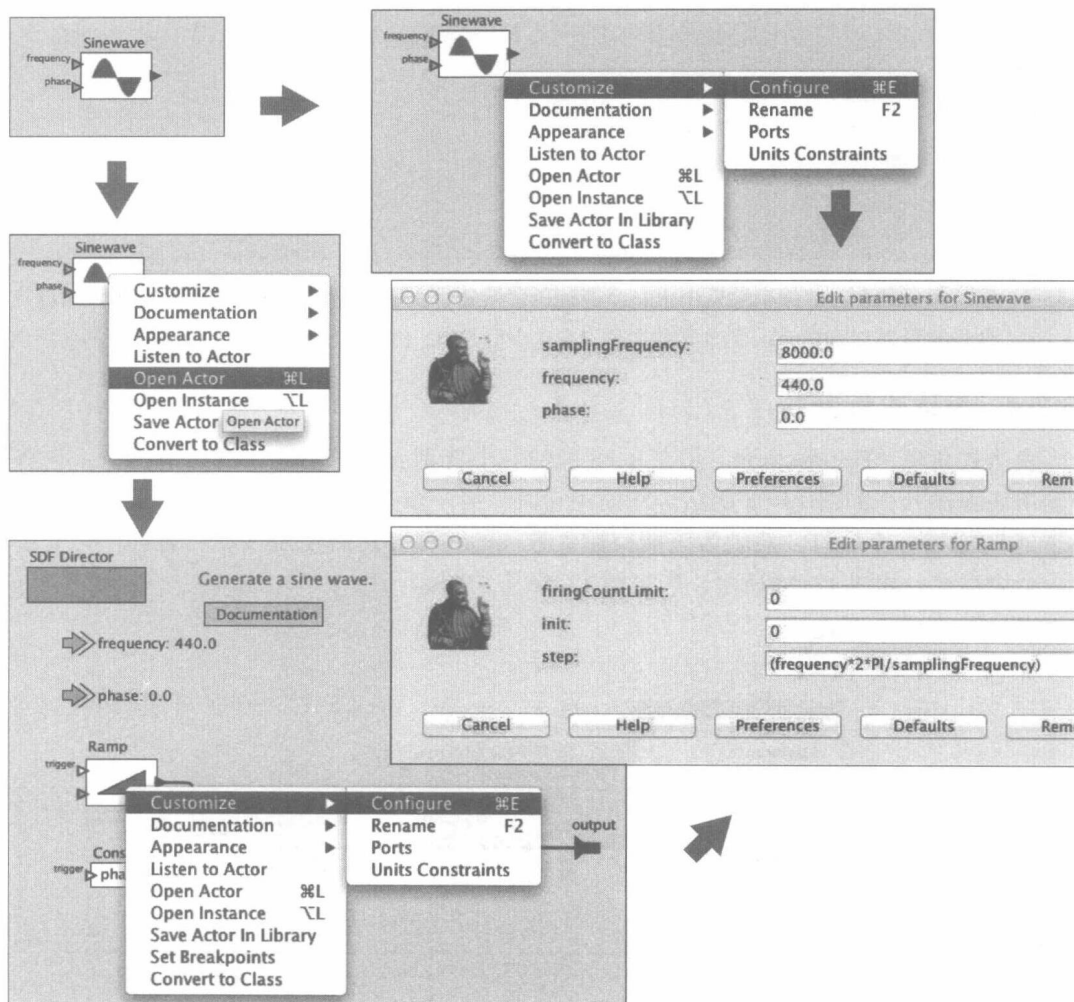


图 13-3 Sinewave 角色的端口参数和它们在分层结构底层的应用

13.2.3 字符串参数

有些参数的值总是字符串。这些参数支持简单的字符串替换机制, 其中字符串的值可以通过使用语法 \$ name 或 \$ {name} (name 是作用域中参数的名字) 的名字来引用作用域中的其他参数。例如, 图 13-4 中的 StringCompare 角色的 firstString 是 “The result is \$PI”。它引用内置常量 PI。secondstring 的值是 “The answer is 3.1415926535898”。如图 13-4 所示, 这两个字符串被认为是等价的, 因为 \$PI 被 PI 的值所替代。

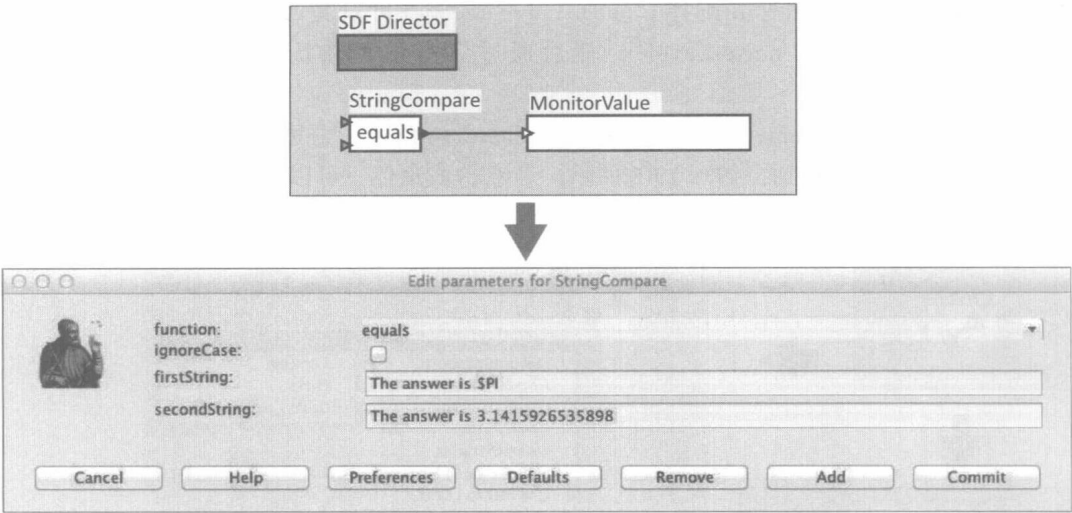


图 13-4 字符串参数在参数编辑器框中由一个淡蓝色背景显示。字符串参数可以包括 \$name 作用域内对变量的引用，其中 name 是变量的名称。在这个例子中，内置常量 \$PI 被第一个参数中的名字引用

13.2.4 表达式角色

表达式角色 (Expression) 是在 Math 中的一个特别有用的角色。默认情况下，它有一个输出而无输入，如图 13-5a 所示。在使用它时，第一步是添加端口，如图 13-5b、c 所示。单击 Add 添加一个端口，然后为该端口输入唯一的名称。然后使用该端口名称作为变量指定一个表达式，如图 13-5d 所示，产生图 13-5e 中的图标。

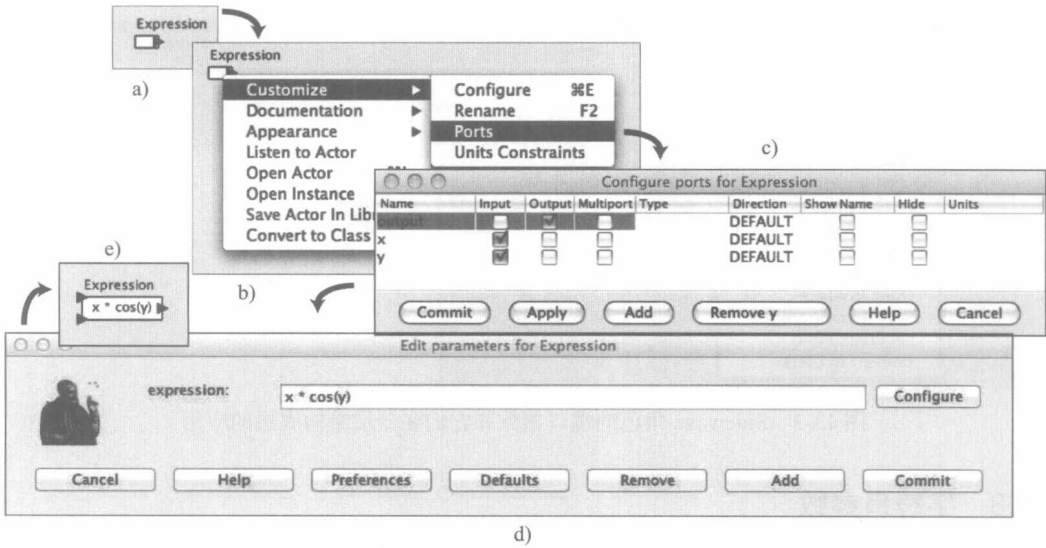


图 13-5 表达式角色 (Expression) 的说明

13.2.5 状态机

表达式给出状态转换的条件、产生输出动作所需要的值，以及在目的状态细化中设置参数值的一些动作。这种机制在前面的章节中介绍过。

13.3 复合数据类型

复合数据类型是一种多数据类型的集合，它聚集了一些其他的数据类型。表达式语言的复合数据类型包括数组、矩阵、记录和联合体。

13.3.1 数组

花括号指定数组中的元素，如 `{1,2,3}` 是整型数组，而 `{"x","y","z"}` 是字符串型数组。这些类型分别可表示为 `arrayType(int, 3)` 和 `arrayType(string, 3)`。一个数组是一个有序列表，所有元素类型一致。如果给数组赋值为混合类型，那么表达式计算器将尝试无损地将转元素转换为一个公共类型。因此，

```
{1, 2.3}
```

将变为

```
{1.0, 2.3}
```

它的类型是 `arrayType(double, 2)`。公共类型可能是 `arrayType(scalar)`，这是一个联合体（可以包含多个不同的类型）。例如，

```
{1, 2.3, true}
```

的值为

```
{1, 2.3, true}
```

值未改变，但是数组的类型现在是 `arrayType(scalar, 3)`。

在图 13-5c 中，“Type”列可用于指定端口的类型。通常没有必要设置端口类型，因为类型推断机制将根据连接来确定其类型（见第 14 章）。然而，有时强制使一个端口有特定的类型是必要的或有用的。

Type 列支持类似于 `arrayType(int)` 的表达式，它指定一个未知长度的数组。然而，若长度是已知的，最好的做法是指定数组长度。要做到这一点，可使用类似 `arrayType(int, n)` 这样的表达式，其中 `n` 是正整数，表示端口预期的数组长度。

数组的元素可以用表达式 `{2*pi, 3*pi}` 赋值。数组可以嵌套。例如，`{{1, 2}, {3, 4, 5}}` 是整数数组的数组。数组中的元素可以如下所示地访问：

```
>> {1.0, 2.3}(1)
2.3
```

注意，索引从 0 开始。当然，如果 `name` 是在一个作用域中变量的名称，且此变量的值是一个数组，则其元素可以类似地访问，如本例：

```
>> x = {1.0, 2.3}
{1.0, 2.3}
>> x(0)
1.0
```

对数组的算术运算是通过对每个元素进行运算实现的，如下所示：

```
>> {1, 2}*{2, 2}
{2, 4}
>> {1, 2}+{2, 2}
{3, 4}
>> {1, 2}-{2, 2}
{-1, 0}
>> {1, 2}^2
{1, 4}
```

```
>> {1, 2}%{2, 2}
{1, 0}
```

此外, 还支持数组的减法、乘法、除法和与标量 (scalar) 数组的求模运算, 如下所示:

```
>> {1.0, 2.0} / 2.0
{0.5, 1.0}
>> 1.0 / {2.0, 4.0}
{0.5, 0.25}
>> 3 * {2, 3}
{6, 9}
>> 12 / {3, 4}
{4, 3}
```

长度为 1 的数组相当于标量, 如下图所示:

```
>> {1.0, 2.0} / {2.0}
{0.5, 1.0}
>> {1.0} / {2.0, 4.0}
{0.5, 0.25}
>> {3} * {2, 3}
{6, 9}
>> {12} / {3, 4}
{4, 3}
```

使用嵌套数组时有显著的差异, 请注意如下的例子:

```
>> {{1.0, 2.0}, {3.0, 1.0}} / {0.5, 2.0}
{{2.0, 4.0}, {1.5, 0.5}}
```

在该例中, 除法的左边参数是有两个元素的数组, 右边参数也是数有两个元素的数组。因此除法是逐个元素的除法。然而, 每个除法是标量数组除法。

检测一个数组与另一个数组是否相等, 可如下所示:

```
>> {1, 2}=={2, 2}
false
>> {1, 2}!={2, 2}
true
```

对于数组其他的比较, 可使用 `compare` 函数 (见表 13-5)。与标量 (scalar) 一样, 检验是否相等使用 `==` 或 `!=` 运算符检验值, 而与类型无关。例如,

```
>> {1, 2}=={1.0, 2.0}
true
```

可通过如下方式得到数组的长度,

```
>> {1, 2, 3}.length()
3
```

可以通过调用 `subarray` 函数提取子数组, 如下所示:

```
>> {1, 2, 3, 4}.subarray(2, 2)
{3, 4}
```

第一个参数是子数组的起始索引, 第二个参数是长度。

也可以用 `extract` 函数提取数组中不连续的元素。这种方法有两种形式。第一种形式采用与原始数组相同长度的布尔数组, 它指出要提取哪些元素, 如下所示:

```
>> {"red", "green", "blue"}.extract({true,false,true})
{"red", "blue"}
```

第二种形式采用一个整数数组, 它给出要提取的索引, 如下所示:

```
>> {"red", "green", "blue"}.extract({2,0,1,1})
{"blue", "red", "green", "green"}
```

可以调用函数 `emptyArray` 来创建一个具有特定元素类型的空数组。例如，为了创建一个空的整数数组，可用：

```
>> emptyArray(int)
{}
```

可以调用 `concatenate` 函数将多个数组组合成一个数组，例如，

```
>> concatenate({1, 2}, {3})
{1, 2, 3}
```

可以调用 `update` 函数更新数组的元素，例如，

```
>> {1, 2, 3}.update(0, 4)
{4, 2, 3}
```

`update` 函数创建一个数组[⊖]。

13.3.2 矩阵

在 Ptolemy II 中，数组是有序的符号集列表。Ptolemy II 也支持矩阵，矩阵较数组显得更专业。目前它们只包含某些基本类型 `boolean`、`complex`、`double`、`fixedpoint`、`int` 以及 `long`，还不支持 `float`、`short` 以及 `unsignedByte` 矩阵。矩阵不能包含任意的数据，例如，它们不能包含矩阵。它们面向的是数据密集型计算。矩阵用方括号表示，逗号分隔行元素，分号分隔行。如 `[1, 2, 3; 4, 5, 5+1]` 给出了一个 2×3 的矩阵（2 行 3 列）。需要注意的是，数组或矩阵元素可以由表达式给出。行向量可以如 `[1, 2, 3]` 这样表示，列向量如 `[1; 2; 3]` 这样表示。可支持某些 MATLAB 风格的数组构造函数。例如，`[1:2:9]` 给出了数组从 1 ~ 9 的奇数元素，等价于矩阵 `[1, 3, 5, 7, 9]`。同样，`[1:2:9; 2:2:10]` 等价于 `[1, 3, 5, 7, 9; 2, 4, 6, 8, 10]`。在语法 `[p:q:r]` 中，`p` 是第一个元素，`q` 是元素之间步长，`r` 是最后一个元素的上限。也就是说，矩阵将不包含大于 `r` 的元素。若指定一个矩阵为混合类型，则元素将被尽可能地转换为公共类型。因此，例如，`[1.0, 1]` 相当于 `[1.0, 1.0]`，但是 `[1.0, 1L]` 是非法的（因为不存在一种允许两元素无损转换的公共类型，见第 14 章）。

使用 `matrixname(n, m)` 函数来引用矩阵中元素，其中 `matrixname` 是作用域中的矩阵变量的名称，`n` 是行索引，`m` 是列索引。索引编号从 0 开始，而不是 1，与 Java 和 MATLAB 类似。例如，

```
>> [1, 2; 3, 4](0,0)
1
>> a = [1, 2; 3, 4]
[1, 2; 3, 4]
>> a(1,1)
4
```

矩阵乘法与数学中的乘法规则一样，如，

```
>> [1, 2; 3, 4]*[2, 2; 2, 2]
[6, 6; 14, 14]
```

当然，如果矩阵的大小不匹配，那么将得到一个错误信息。为了实现逐个元素相乘，使

⊖ 事实上，`update` 函数创建一个新令牌，其类型是 `UpdatedArrayToken`，它在令牌中追踪更新元素同时保存未改变的元素。另一种方法是产生一个新的 `ArrayToken`，这将导致分配内存并复制整个源数组。

用 `multiplyElements` 函数（见表 13-8）。矩阵加法和减法都是逐个元素进行的，与期望的一样，但不支持除法运算符。逐个元素相除可以调用 `divideElements` 函数，矩阵求逆的乘法，可以调用 `inverse` 函数（见表 13-8）。矩阵可以进行类型为 `int`、`short` 或者 `unsignedByte` 的幂运算，这相当于它自身相乘一定的次数。例如，

```
>> [3, 0; 0, 3]^3
[27, 0; 0, 27]
```

一个矩阵，也可以乘以或除以一个标量，如下所示：

```
>> [3, 0; 0, 3]*3
[9, 0; 0, 9]
```

一个矩阵可以加上一个标量。它还可以减去一个标量，或者一个标量减去它。例如，

```
>> 1-[3, 0; 0, 3]
[-2, 1; 1, -2]
```

可以检测一个矩阵是否与另一个矩阵相等，如下所示：

```
>> [3, 0; 0, 3] != [3, 0; 0, 6]
true
>> [3, 0; 0, 3] == [3, 0; 0, 3]
true
```

对于其他的矩阵比较，使用 `compare` 函数（见表 13-7）。与标量一样，使用 `==` 或 `!=` 运算符检测值是否相等，与类型无关。例如，

```
>> [1, 2] == [1.0, 2.0]
true
```

为了对特定类型的相等性进行检测，使用 `equals` 方法，如下例子所示：

```
>> [1, 2].equals([1.0, 2.0])
false
>> [1.0, 2.0].equals([1.0, 2.0])
true
```

13.3.3 记录

记录类型的数据是包含命名字段的复合类型，其中每个字段都有值。每个字段的值可有不同的类型。记录用花括号分隔，每个字段有一个名称。例如，`{a=1, b="foo"}` 是有两个字段的记录，分别命名为“a”和“b”，其值分别为 1（整数）和“foo”（字符串）。一个字段的主键可以是任意字符串，同样能够被引用。只有与合法的 Java 标识符一样的字符串可以不带引号。需要注意的是，带引号的字符串内的引号必须使用反斜杠转义。一个字段的值可以是任意的表达式，记录可以嵌套（一个记录令牌的字段的可能是另一个记录令牌）。

有序记录与普通记录相似，只是它保存了标记的原始顺序。有序记录使用方括号分隔而不是花括号。例如，`[b="foo", a=1]` 是有序记录，其中“b”是第一个标记。

合法的 Java 标识符的字段可以使用点（.）运算符访问，如果是函数调用则需要用括号。例如，下面的两个表达式：

```
{a=1,b=2}.a
{a=1,b=2}.a()
```

都是得到 1。

另一种访问字段的语法是使用 `get()` 方法。注意，如果要求使用引号时，这是访问字段的唯一的方法。例如：

```
{ " a"=1, "\"b"=2 }.get("\"b")
```

得到 2。

算术运算符 `+`、`-`、`*`、`/` 和 `%` 可应用于记录。如果记录没有相同的字段，那么运算仅应用于匹配的字段，其结果仅包含匹配的字段。例如，

```
{foodCost=40, hotelCost=100}
+ {foodCost=20, taxiCost=20}
```

产生的结果为，

```
{foodCost=60}
```

考虑交运算，这里运算指定如何合并相交字段的值。也可以形成一个交集而没有相应的运算。在这种情况下，使用 `intersect` 函数，则形成一个只包含两个指定记录的公有字段的新记录，其值为第一个记录中的值。例如，

```
>> intersect({a=1, c=2}, {a=3, b=4})
{a=1}
```

记录通过调用 `merge` 直接连接。这个函数带有两个参数，即两个记录令牌。如果两个记录令牌有相同的字段，那么使用第一个记录中的字段值。例如，

```
merge({a=1, b=2}, {a=3, c=3})
```

产生结果 `{a=1, b=2, c=3}`。

记录可以进行比较，如下例所示：

```
>> {a=1, b=2}!={a=1, b=2}
false
>> {a=1, b=2}!={a=1, c=2}
true
```

注意，两个记录相等，当且仅当它们有相同的字段标记和值匹配。与标量一样，值匹配与类型无关。例如：

```
>> {a=1, b=2}=={a=1.0, b=2.0+0.0i}
true
```

对于普通（无序的）记录，字段的顺序无关紧要。因此，

```
>> {a=1, b=2}=={b=2, a=1}
true
```

此外，普通记录字段是按字母顺序排列的，与它们定义的顺序无关。例如，

```
>> {b=2, a=1}
{a=1, b=2}
```

有序记录的相等性比较遵循字段的原始顺序。例如，

```
>> [a=1, b=2]==[b=2, a=1]
false
```

另外，有序记录字段总是按照它们定义的顺序排列。例如，

```
>> [b=2, a=1]
[b=2, a=1]
```

为进行特定类型的相等测试，使用 `equals` 方法，如下所示，

```
>> {a=1, b=2}.equals({a=1.0, b=2.0+0.0i})
false
>> {a=1, b=2}.equals({b=2, a=1})
true
```

最后，可以使用空记录函数 (`emptyRecord`) 创建一个空记录：

```
>> emptyRecord()
{}
```

13.3.4 联合体

可能有多个不同的数据类型需要通过同一连接发送的情况，或者一个变量可能通过多个数据类型中的一个值呈现。Ptolemy II 提供**联合体**来满足这种需要。联合体定义如下：

```
{| a = int, b = complex |}
```

这表明是一个整型或复数类型的端口或变量。联合体的一个典型应用是使用 `UnionMerge` 和 `UnionDisassembler` 角色。

例 13.1 如图 13-6 所示，这是一个由两个数据源 (`DiscreteClock` 和 `PoissonClock`) 组成的 DE 模型，其中 `DiscreteClock` 产生 `int` 型的输出，而 `PoissonClock` 产生 `boolean` 型的输出，这两个数据流由 `UnionMerge` 组合而成，它的输出类型就变成了一个联合体。联合体中类型的名称在构建模型时由 `UnionMerge` 输入端口的名称决定并添加进来的。下方的 `Display` 角色显示合并后的流，表明每个显示的令牌只有一种类型：整型或布尔型。

沿着上面的路径，`UnionDisassembler` 角色用来从流中提取“b”类型，只有“b”才能通过到输出，并被判断是否为 `boolean` 型。

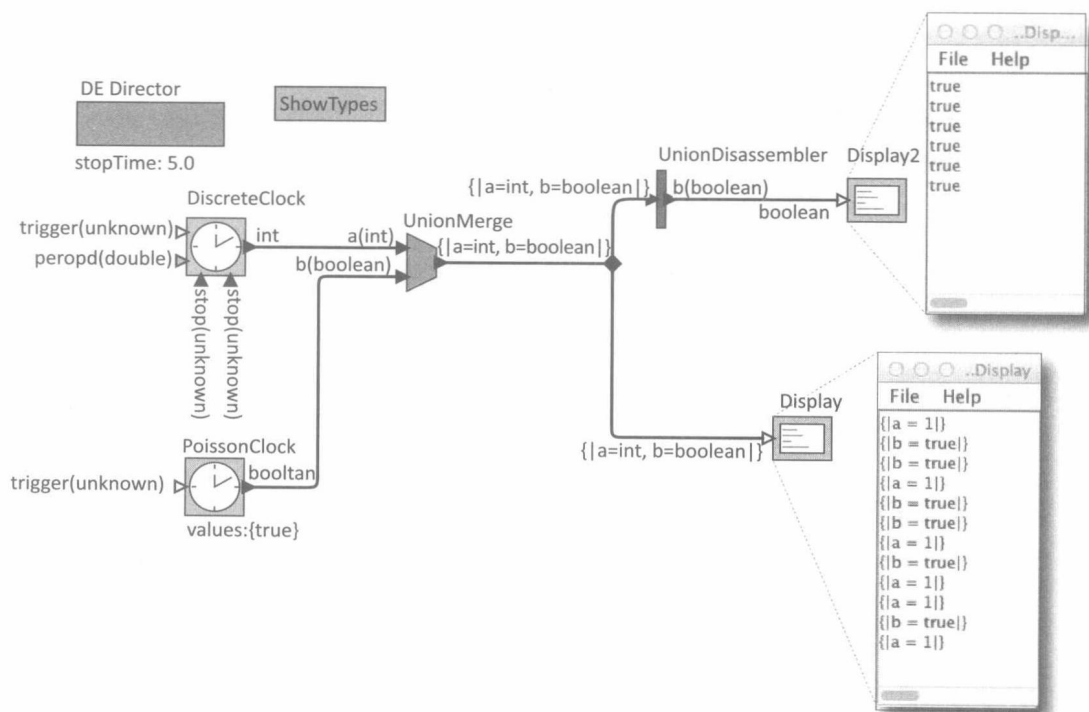


图 13-6 联合体允许定义一个联合体来包含多个类型

13.4 令牌运算

表达式中的每个元素和子表达式都表示 Ptolemy II 中令牌类的一个实例（也有可能从 Token 类派生出一个类）。表达式语言支持多种访问底层 Java 代码令牌的运算。

13.4.1 调用方法

表达式语言支持多种给定令牌的调用方法，只要方法的参数为 Token 类型，返回类型是令牌（或从令牌派生的类，或能被表达式分析程序转换为令牌的某些类型，如 string、double、int 等）。它的语法为 (token.methodName(args))，其中 methodName 是方法的名称，args 是逗号分隔的参数集。每个参数本身可以是一个表达式。注意 token 周围的括号不是必需的，仅为方便用户查看。ArrayToken 和 RecordToken 类有一个长度计算 (length) 方法，如下例所示：

```
{1, 2, 3}.length()
{a=1, b=2, c=3}.length()
```

两者都返回整数 3。

MatrixToken 类有 3 个特别实用的方法，如下所示：

```
[1, 2; 3, 4; 5, 6].getRowCount()
```

返回 3，

```
[1, 2; 3, 4; 5, 6].getColumnCount()
```

返回 2，

```
[1, 2; 3, 4; 5, 6].toArray()
```

返回 1、2、3、4、5、6。后面的函数可以使用 MATLAB 风格的语法来创建数组。例如，为了获得一个从 1 ~ 100 的整数数组，用户可以输入：

```
[1:1:100].toArray()
```

13.4.2 访问模型元素

表达式中的模型可以引用元素模型并可以采用调用方法。表达式对参数赋值可以引用参数容器中的任何对象。

例 13.2 图 13-7 中的模型带有 4 个参数：P, 1 ~ P, 4。

P, 1 的表达式为：

```
Const2.value
```

其中，Const2 引用名为“Const2”的角色，它包含在 P1 容器中的，因此 Const2.value 引用角色 Const2 的值参数。因此，参数 P1 的值等于 Const2 的值参数的值。

参数表达式中的关键字 this 引用包含参数的对象。

例 13.3 在图 13-7 中，Const2 有一个带有表达式的 value 参数（如图标所示）：

```
this.getName() + ": " + P2
```

其中，this 指的是 Const2，故 this.getName() 返回一个字符串，该字符串的名称为“Const2”。表达式的其他部分执行字符串连接运算，附加一个冒号和值参数 P2。

参数 P2 的表达式为：


```
this.entityList().size()
```

在这种情况下，`this` 引用顶层模型的 `P2` 的容器。因此，`this.entityList()` 返回一个顶层模型包含的实体（角色）列表。最后，`this.entityList().size()` 返回包含在顶层模型中的角色数量为 5。

第二个输出是 `P2` 的值（即 5），`P2` 的前面是产生输出的 `Const` 角色的名字和一个冒号。第一个输出是值 `P1`（即字符串 `"Const2:5"`），`P1` 的前面是产生输出的 `Const` 角色的名字和一个冒号。

因此该模型的前两个输出很容易理解：

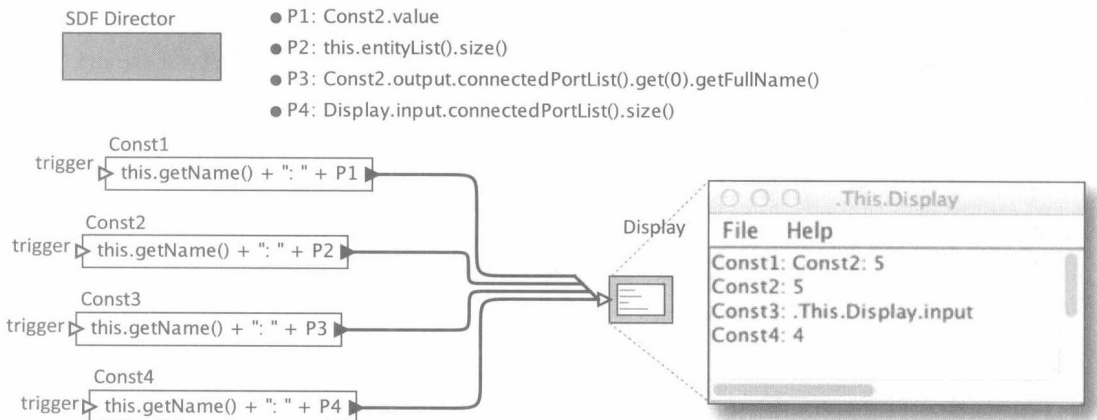


图 13-7 如图所示，表达式可以访问模型的元素

```
Const1: Const2: 5  
Const2: 5
```

参数可利用调用方式来衔接模型中的整个连接，如下所示。

例 13.4 在图 13-7 中，参数 `P3` 的表达式为：

```
Const2.output.connectedPortList().get(0)  
.getFullName()
```

这样就能得到端口列表中第一个端口的全名，该端口列表与角色 `Const2` 的 `output`（输出）端口相连。具体来说，它将返回字符串 `".This.Display.input"`，并由角色 `Const3` 显示。

同样，`P4` 表达式为：

```
Display.input.connectedPortList().size()
```

它返回连接到 `Display` 输入端口的源的数量。

角色、参数和端口调用方法概述请参见第 12 章。为了得到完整的清单，可参阅 `Ptolemy II` 的代码文档。

13.4.3 类型分配

`cast` 函数可以用来显式地将一个值赋给一个类型，当以 `cast(type, value)` 形式被调用给值分配类型时（`type` 是目标类型，`value` 是要被赋予的值），返回一个指定类型的新值（如果一个预先定义的类型是可用的）。例如，`cast(long, 1)` 返回 `1L`，其值等于 1，但它是一个长整型类型；`cast(string, 1)` 返回 `"1"`，它是字符串数据类型。

13.4.4 函数定义

用户能够通过表达式语言定义新的函数，语法如下：

```
function(arg1:Type, arg2:Type...)
  function body
```

function 是定义函数的关键字。可以不指明参数的类型，在该情况下，表达式语言会进行推断。函数体给出定义函数返回值的表达式。返回类型总是根据参数类型和表达式来推断的。例如：

```
function(x:double) x*5.0
```

定义了一个带有 `double` 类型参数的函数，它乘以 5.0，返回 `double` 类型。上面表达式的返回值是函数本身。因此，表达式计算器产生：

```
>> function(x:double) x*5.0
(function(x:double) (x*5.0))
```

为了将函数作为一个参数，只需要

```
>> (function(x:double) x*5.0) (10.0)
50.0
```

另外，在表达式计算中，可以将函数赋给一个变量，然后使用变量名来应用该函数。例如，

```
>> f = function(x:double) x*5.0
(function(x:double) (x*5.0))
>> f(10)
50.0
```

13.4.5 高阶函数

函数可以作为参数传递给已经定义的某些高阶函数（见表 13-15）。例如，`iterate` 函数有 3 个参数：一个函数、一个整数和一个应用函数的初始值。它首先将初始值应用到函数中，然后应用到程序中，再应用到结果中，将收集的结果放入数组中，该数组的长度由第二个参数给定。例如，为了得到一个其值乘以 3 的数组，尝试如下操作：

```
>> iterate(function(x:int) x+3, 5, 0)
{0, 3, 6, 9, 12}
```

作为参数该函数给它的参数加 3。结果是指定的初始 `value(0)` 加上 3，再以这个结果作为初始值 `value(0)` 进行迭代。

另一个实用的高阶函数是 `map` 函数。它有一个函数和一个数组作为参数，并简单地将函数作用于数组的各个元素以得到一个重新构造的数组。例如，

```
>> map(function(x:int) x+3, {0, 2, 3})
{3, 5, 6}
```

Ptolemy II 还支持折叠函数（`fold`），表达式可将该函数用于编写循环程序。`fold` 函数将函数作用于数组的每个元素，并累计结果。基于数组的 `fold` 函数有两个参数，当前的累积结果和一个数组元素。当 `fold` 函数应用于数组的第一个元素时，累计结果就是初始值。

例 13.5

```
fold(
```

```
function(x:int, e:int) x + 1,
0, {1, 2, 3}
)
```

计算数组 {1, 2, 3} 的长度, 结果是 3, 相当于 {1, 2, 3}.length()。具体来说, 要折叠的函数是 `function(x:int, e:int)x+1`。如果给定参数 `x` 和 `e`, 则返回 `x+1`, 此处忽略了二个参数 `e`。首先将它应用于初始值 0 和数组的第一个元素 1, 得到 1。然后首先将累计结果 1 和 1 相加得到 2。它的调用次数与数组 {1, 2, 3} 中的元素个数相同。因此, `x` 从起始值 0 增加 3 次, 最后得到了。

例 13.6 下面的变体是不忽略数组元素值的例子:

```
fold(
function(x:int, e:int) x + e,
0, {1, 2, 3}
)
```

它计算数组 {1, 2, 3} 中所有元素的和, 得到 6。

例 13.7

```
fold(
function(x:arrayType(int), e:int)
e % 2 == 0 ? x : x.append({e}),
{}, {1, 2, 3, 4, 5}
)
```

它计算只包含奇数的数组 {1, 2, 3, 4, 5} 的子数组。结果是 {1, 3, 5}。

例 13.8 令 `C` 是一个角色。

```
fold(
function(list:arrayType(string),
port:object("ptolemy.kernel.Port"))
port.connectedPortList().isEmpty() ?
list.append({port}) : list,
{}, C.portList()
)
```

它返回一个 `C` 的端口的列表, 这些端口不与任何其他端口相连接 (`connectedPortList()` 为空)。返回列表的每个端口都封装在 `ObjectToken` 上。

13.4.6 模型中的函数调用

在 `Ptolemy II` 模型中, 函数的典型应用是在模型中定义一个参数, 它的值是一个函数。假设名为 `f` 的参数值为:

```
function(x:double)x*5.0
```

然后在这个参数的作用域内, 表达式 `f(10.0)` 将产生结果 50.0。在 `PtolemyII` 模型中函数也能够随着连接传递。

例 13.9 考虑 13-8 中的模型, 该例中, `Const` 角色定义了一个函数, 该函数只是对参数做平方运算。因此, 它的输出是一个带有类型函数的令牌。令牌传送到 `Expression` 角色的“`f`”输入端。`Expression` 角色通过将它包含的这个函数, 应用到“`y`”输入端提供的令牌, 而该令牌由 `Ramp` 角色提供, 因此结果如图 13-8 右图中的曲线所示。

例 13.10 一个更复杂的使用示例如图 13-9 所示。在这个例子中, `Const` 角色产生一个

函数，然后 Expression 角色使用它来创建新的函数，然后 Expression2 使用新函数来执行计算。这里执行的计算是将 Ramp 输出乘以 Ramp 输出的平方，从而实现立方的计算。

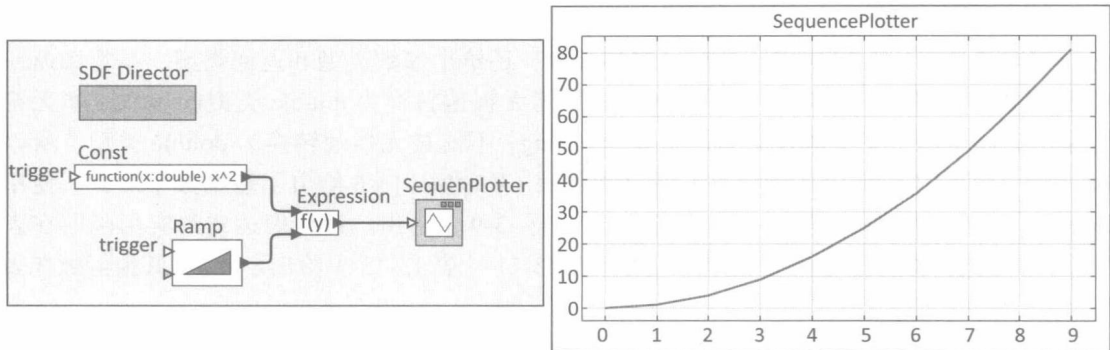


图 13-8 函数从一个角色传递给另一个角色

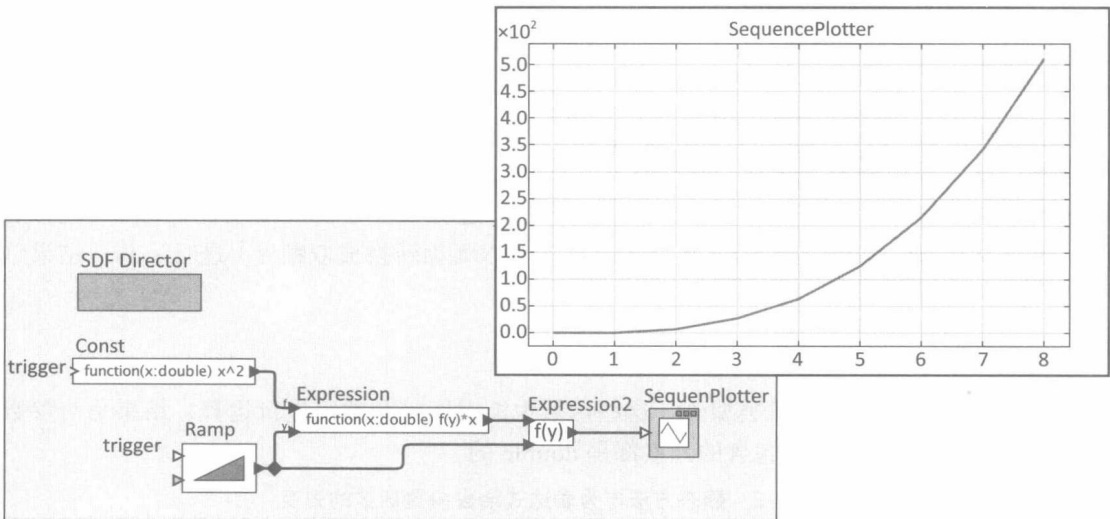


图 13-9 角色之间传递函数的复杂例子

13.4.7 递归函数

函数可以进行递归，如下例（比较难理解）所示：

```
>> fact = function(x:int,f:(function(x,f) int)) (x<1?1:x*f(x-1,f))
(function(x:int, f:function(a0:general, a1:general) int)
  (x<1)?1:(x*f((x-1), f)))
>> factorial = function(x:int) fact(x,fact)
(function(x:int) (function(x:int, f:function(a0:general, a1:general) int)
  (x<1)?1:(x*f((x-1), f)))(x, (function(x:int,
  f:function(a0:general, a1:general) int) (x<1)?1:(x*f((x-1), f)))))
>> map(factorial, [1:1:5].toArray())
{1, 2, 6, 24, 120}
```

第一个表达式定义了一个名为“fact”的函数，它将一个函数作为参数，如果参数大于或等于 1，则递归使用该函数。第二个表达式使用函数“fact”定义一个新函数“factorial”。最后一条命令将函数“factorial”应用于数组来计算阶乘。

13.4.8 内置函数

表达式语言包括了一组如 `sin`、`cos` 等这样的函数。这些函数是内置的，它包括图 13-2 中类似所有的静态方法[⊖]，它们一起为表达式语言提供了一个丰富的指令集[⊖]。

在本章最后一节中列出了当前可用的函数，还给出参数类型和返回类型。参数和返回类型的种类众多，例如，`acos` 可以使用任何能无损地转换为 `double` 类型的参数，如无符号字节、短整型、整型和浮点型。长整型 (`long`) 不能被无损地转换为 `double` 类型，所以 `acos(1L)` 会失败。表 13-4 给出了三角函数。表 13-5 和表 13-6 给出了基本数学函数。使用或返回矩阵、数组或记录的函数在表 13-7 ~ 表 13-9 中给出。评估表达式的实用函数在表 13-10 中给出。执行信号处理运算的函数在表 13-11 ~ 表 13-13 中给出。I/O 和其他函数在表 13-15 和表 13-16 中给出。

在大多数情况下，对标量参数进行运算的函数也可以对数组和矩阵进行运算。因此，可以使用如下表达式用正弦波填充一个行向量，

```
sin([0.0:PI/100:1.0])
```

也可以构造一个如下的数组：

```
sin({0.0, 0.1, 0.2, 0.3})
```

对 `double` 类型进行运算的函数通常也可以对 `int`、`short` 或 `unsigned Byte` 进行运算，因为这些可以无损转换为 `double`，但一般不能对 `long` 或 `complex` 进行运算。可用的函数表如附录所示。例如，表 13-4 显示了三角函数。注意，这些可以对 `double` 或者 `complex` 进行运算，也可以对 `int`、`short` 和 `unsigned Byte`（这些可以无损转换成双精度）进行运算，结果总是 `double`。例如，

```
>> cos(0)
1.0
```

除了表 13-2 所示的标量类型外，这些函数也能对矩阵和数组进行运算，结果是与参数大小相同的矩阵或数组，但包含的元素都是 `double` 的。

表 13-2 静态方法作为表达式语言内置函数的列表

<code>java.lang.Math</code>	<code>ptolemy.math.IntegerMatrixMath</code>
<code>java.lang.Double</code>	<code>ptolemy.math.DoubleMatrixMath</code>
<code>java.lang.Integer</code>	<code>ptolemy.math.ComplexMatrixMath</code>
<code>java.lang.Long</code>	<code>ptolemy.math.LongMatrixMath</code>
<code>java.lang.String</code>	<code>ptolemy.math.IntegerArrayMath</code>
<code>ptolemy.data.MatrixToken.</code>	<code>ptolemy.math.DoubleArrayStat</code>
<code>ptolemy.data.RecordToken.</code>	<code>ptolemy.math.ComplexArrayMath</code>
<code>ptolemy.data.expr.UtilityFunctions</code>	<code>ptolemy.math.LongArrayMath</code>
<code>ptolemy.data.expr.FixPointFunctions</code>	<code>ptolemy.math.SignalProcessing</code>
<code>ptolemy.math.Complex</code>	<code>ptolemy.math.FixPoint</code>
<code>ptolemy.math.ExtendedMath</code>	<code>ptolemy.data.ObjectToken</code>

⊖ 需要注意的是，例如 `String.format()` 这类被称为方法的函数，带有一个 `Object[]` 参数是很困难的，因为描述一个 Java 目标的数组比较麻烦。例如用 `java.lang.Double` 来替代 `ptolemy.data.type.DoubleToken`。

⊖ 然而，当你在通过注册一个包含静态方法的类来编写 Java 代码的时候，有一系列的方法可以被轻松的用来扩展（详见 `ptolemy.data.exper` 包中的 `PtParser` 类）。

表 13-7 显示了除了三角函数之外的算术函数。与三角函数一样，对 `double` 进行运算的函数也能对 `int`、`short` 和 `unsigned Byte` 型进行运算，除非指定，否则将返回与参数为 `double` 一样的结果。表 13-2 中这些带有标量参数的函数，也能对矩阵和数组进行运算。例如，因为该表表明 `max` 函数可以采取 `(int, int)` 作为参数，所以它也可以用 `(int, int)` 作为参数。

```
>> max({1, 2}, {2, 1})
{2, 2}
```

注意，表 13-2 表明 `max` 可以带有 `int` 参数，例如：

```
>> max({1, 2, 3})
3
```

在前一种情况下，函数逐个应用到两个参数。在后一种的情况下，返回值为单一参数的最大值。

表 13-7 只显示了使用矩阵、数组或记录（也就是说，没有相应的标量运算）的函数。回想一下，大多数对标量运算的函数也能对数组和矩阵进行运算。表 13-10 显示了评估作为字符串的表达式或表示数字为字符串的实用函数。其中，`eval` 函数是最灵活的。

还有一些函数具有细微的属性，需要进一步说明，解释如下。

1. `eval()` 和 `traceEvaluation()`

内置函数 `eval` 像表达式语言中的表达式一样评估一个字符串。例如，

```
eval("[1.0, 2.0; 3.0, 4.0]")
```

返回一个 `double` 的矩阵。下面的组合函数可以用来从文件中读取参数：

```
eval(readFile("filename"))
```

文件名与当前工作路径（`Ptolemy II` 启动，路径同 Java 虚拟机属性 `user.dir` 提供的一致）、用户主目录（属性 `user.home` 提供）或者类路径相关，包括 `Ptolemy II` 安装的路径。注意，如果在 `Expression` 角色中使用 `eval`，函数将不能为类型系统推断任何更具体的输出类型。如果需要更具体的输出类型，则需要给 `eval` 的结果分配类型。例如，将它强制转换为 `double`：

```
>> cast(double, eval("pi/2"))
1.5707963267949
```

`traceEvaluation` 函数评估作为字符串的表达式，与 `eval` 函数很类似，但不是报告结果，而是准确地报告如何评估表达式进行计算。这可以用来进行表达式调试，特别是在表达式语言被用户扩展的情况下。

2. `random()` 和 `gaussian()`

表 13-5 和表 13-6 的 `random` 和 `gaussian` 函数返回一个或多个随机数。用最小数量的参数（分别为零个或两个）返回一个数字。加上一个额外参数，返回一个指定长度的数组。再加上另一个额外参数，则返回指定行数和列数的矩阵。

使用 `Ptolemy II` 上的这些函数时需要注意一个关键细节。只有当表达式被计算时包含它的表达式才能进行计算，表达式的结果可反复使用而无需对表达式重新计算。因此，如果 `Const` 角色的值参数设置为 `random()`，那么它的输出将是一个随机常数，即，每次点火时它都不会改变。然而，在模型连续运行时，输出将改变。相反，如果它应用在 `Expression` 角色中，那么每次点火将进行一次表达式计算，因此产生一个新的随机数。

3. property()

property 函数通过名字可访问 Java 虚拟机系统属性。比较有用的系统属性是：

- ptolomy.ptII.dir: Ptolemy II 的安装目录。
- ptolomy.ptII.dirAsURL: Ptolemy II 的安装目录，得到一个 URL。
- user.dir: 当前的工作目录，它是启动当前可执行程序的目录。

要获得 Java 虚拟机属性的完整列表，请参考 `ava.lang.System.getProperties` 的 Java 文档。

4. remainder()

这个函数对两个 IEEE 754 标准规定的参数计算余数，它与由 `%` 运算符计算的模运算不同。`remainder(x, y)` 的结果是 $x - yn$ ，其中 n 是最接近准确值 x/y 的整数。如果两个整数相当接近，那么 n 将是偶整数。产生的结果可能令人惊讶，如下例所示：

```
>> remainder(1,2)
1.0
>> remainder(3,2)
-1.0
```

将它与下面进行比较

```
>> 3%2
1
```

它们有两点不同。数值结果不同：一个是 `int` 型，而 `remainder` 总是产生 `double` 型的结果。`remainder()` 函数由 `java.lang.Math` 类来实现，称为 `IEEERemainder()`。这些类的文档给出以下特殊情况：

- 如果参数是 NaN，或者第一个参数是无限的，或者第二个参数是正 0 或者负 0，那么结果是 NaN。
- 如果第一个参数是有限的，第二个参数是无限的，那么结果与第一参数相同。

5. DCT() 和 IDCT()

离散余弦变换 (Discrete Cosine Transform, DCT) 函数可以有 1 个、2 个或 3 个参数。在所有三种情况下，第一个参数是长度 $N > 0$ 的数组，且 DCT 返回

$$X_k = s_k \sum_{n=0}^{N-1} x_n \cos \left[(2n+1)k \frac{\pi}{2D} \right] \quad (13-1)$$

k 为 $0 \sim D-1$ ， N 是指定数组的长度， D 是 DCT 的大小。如果只给出一个参数，那么设置 D 与两个比 N 大的数相等。如果给出第二个参数，那么它的值是 DCT 的阶 (order)，DCT 的大小是 2^{order} 。如果给出第三个参数，那么它根据下表指定比例因子 s_k ：

如果未给出第三个参数，默认是“标准的”。IDCT 函数与此类似，它也可以有 1 个、2 个或者 3 个参数。在这种情况下，公式是

$$x_n = \sum_{k=0}^{N-1} s_k X_k \cos \left[(2n+1)k \frac{\pi}{2D} \right] \quad (13-2)$$

表 13-3 DCT 函数的标准化选项

名称	第三个参数	数据标准化
标准的	0	$s_k = \begin{cases} \frac{1}{\sqrt{2}} & k=0 \\ 1 & \text{其他} \end{cases}$
非标准的	1	$s_k=1$
规格化正交的	2	$s_k = \begin{cases} \frac{1}{\sqrt{D}} & k=0 \\ \sqrt{\frac{2}{D}} & \text{其他} \end{cases}$

13.5 空值令牌

在类似于 R 和 SAS 的分析系统，空令牌或令牌丢失是非常常见的，它们用于处理数据稀少的数据源。根据数据库术语，令牌丢失有时也称作空令牌（null token）。由于 null 是 Java 的关键字，所以本书使用术语空值（nil）。在分析不需要时刻测量的真实世界数据（比如温度）时，空值令牌（nil token）是非常有用的。原则上，人们希望一个不需要得到所有数据的 TolerantAverage 角色——当 TolerantAverage 角色检测到一个空值令牌时它就忽略掉它。注意，这可能导致不确定性。例如，如果预计 cwerage 有 30 个值，而其中 29 个都是空值令牌，那么平均值就不是很准确。

在 Ptolemy II 中，如果所有参数是空值令牌，那么对令牌的运算就产生一个空值令牌。因此，Average 角色就与 TolerantAverage 角色不一样。当接收一个空值令牌时，其后的结果都将是空值令牌。当运算产生一个空值令牌时，该空值令牌的类型就会与运算产生的类型一样，因此，类型安全得以保障。然而，不是所有的数据类型都支持空值令牌。比如，各种矩阵类型没有空值，因为基本矩阵是 Java 自带类型，它不支持空值矩阵。

表达式语言定义了一个名为 nil 的常量，它的值为零，类型为 niltype（空值类型）（见第 14 章）。表达式语言函数 cast 可以生成其他类型的空值。例如，“cast(int, nil)”将返回一个值为空值的整型令牌。

13.6 定点数

Ptolemy II 包括一个定点数据类型。在表达式语言中，一般用以下格式来表示一个定点值：

```
fix(value, totalBits, integerBits)
```

因此，假设有一个定点值为 5.375，使用 8 位精度来表示它，其中 4 位表示（有符号的）整数部分，那么这个定点数可以表示为：

```
fix(5.375, 8, 4)
```

这个值也可以是一个 double 矩阵。将这个值做四舍五入处理，产生一个具有指定精度的近似值。如果这个近似值超出了范围，那么它是饱和的，说明返回最大或最小的定点值，取决于指定值的符号。例如，

```
fix(5.375, 8, 3)
```

产生 3.968758，最大值精度可能为（8/3）。

除了 fix 函数外，表达式语言还提供了一个 quantize 函数。参数与 fix 函数的参数一样，但其返回类型为 DoubleToken 或 DoubleMatrixToken 而不是 FixMatrix 或 FixMatrixToken。因此，这个函数可用于量化双精度值，而显式地用定点表示。

为了在表达语言中使用定点令牌（FixToken），可使用下列函数：

- 使用表达式语言创建一个定点令牌：

```
fix(5.34, 10, 4)
```

这个函数创建一个定点令牌。在这种情况下，尝试将 5.34 表示成 10 位精度，其中 4 位表示整数部分。有可能导致量化错误，因此采用默认的近似量化。

- 使用表达式语言创建一个具有定点值的矩阵：

```
fix([ -.040609, -.001628, .17853 ], 10, 2)
```

这个函数将产生一个 1 行 3 列的定点数矩阵令牌 (FixMatrixToken)，其中每个元素都是精度为 (10/2) 的定点值。产生的 FixMatrixToken 将给定的双精度矩阵的每个元素都转换成 10 位精度，其中 2 位表示整数部分。在默认情况下使用了近似量化。

- 使用表达式语言创建一个 DoubleToken，它是给定的 double 值的量化版本：

```
quantize(5.34, 10, 4)
```

这个函数将产生一个 DoubleToken。产生的 DoubleToken 包含由 5.34 转换的 10 位精度表示（其中 4 位表示整数部分）的 double 类型值。该做法可能产生完全量化错误，因此默认情况下使用近似量化。

- 使用表达式语言创建一个量化为特定精度的 double 型矩阵：

```
quantize([ -.040609, -.001628, .17853 ], 10, 2)
```

这个函数将产生一个 1 行 3 列的 DoubleMatrixToken。通过将给定的矩阵转换为 10 位精度表示（其中 2 位表示整数部分）来获得令牌的元素。这些值都转换回双精度型，而不是定点值，在默认情况下使用了近似量化。

13.7 单位

Ptolemy II 支持单位系统 (unit system)，它建立在表达式语言之上。单位系统允许参数值用单位表达，如 “1.0 *cm” 等同于 “0.01 *meters”。这样表示 (* 表示乘法) 是因为 “cm” 和 “meters” 实际上是变量，当在一个模型中使用单位系统后它就变成了作用域。实用程序库中提供一些简单的单位系统（主要是作为例子）。

图 13-10 展示了一个使用简单单位系统的模型。这个简单单位系统称为 BasicUnits (基本单位)。可以通过双击它的图标来查看它定义的单位，或者通过调用 “Customize”|“Configure” 来查看，如图 13-11 所示。在图 13-11 中可以看到 “meters”、“meter” 和 “m” 的定义，它们的含义相同。此外，定义了 “cm”，给定了值 “0.01 *meters”，定义了 “in”、“inch” 和 “inches”，并给定了值 “2.54 *cm”。

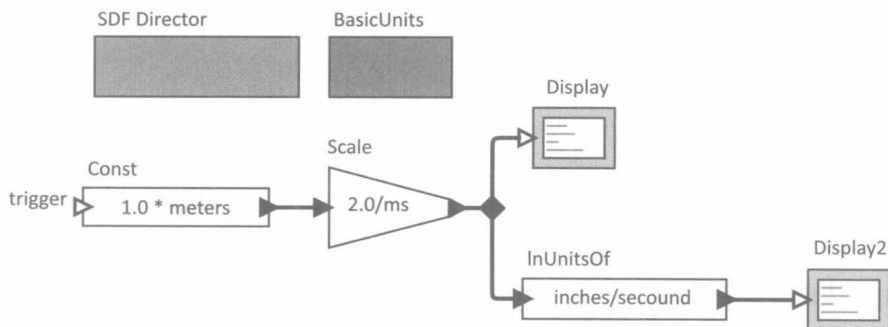


图 13-10 包括单位系统的模型示例

在图 13-10 的例子中，将值为 “1.0 * meter” 的常数送入比例因子为 “2.0 /ms” 的 Scale 角色。随着时间推移它产生一个有长度维度的结果。如果将这个结果直接送入 Display

角色，则它显示为“2000.0meters/seconds”，如图 13-12 的顶层所示。长度标准单位是米，时间标准单位是秒。

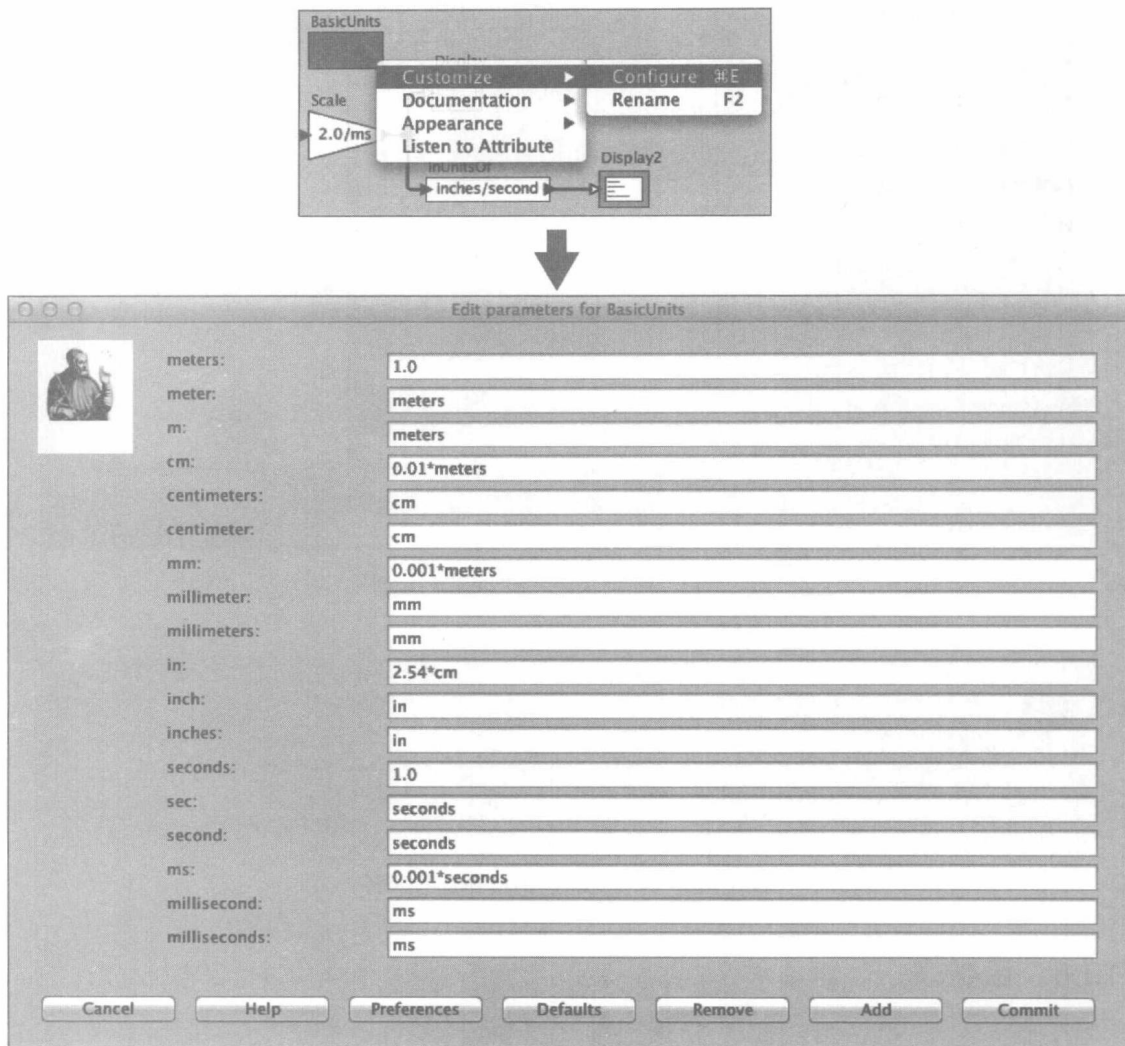


图 13-11 单位系统中定义的单位可以通过双击或右击并选择 Customize 和 Configure 来检查

在图 13-10 中，还需要将结果送入 InUnitsOf 角色，它的输入除以它的参数，检查并确保它的输出是没有单位的。也就是说，2 meters/ms 大约等于 78.740 inches/second。

InUnitsOf 角色可用于确保在模型中正确解释数据，它可以有效地捕获某些关键错误。例如，在图 13-10 中，如果我们将“seconds/inch”而不是“inches/second”送入 InUnitsOf 角色，就会得到图 13-13 中的异常，而不是图 13-12 中的执行情况。

单位系统完全建立在 Ptolemy II 表达式语言的基础上。单位系统图标实际上表示作用域扩展属性的实例，这些属性是作用域内参数的属性，就好像这些参数直接包含在作用域扩展属性中。也就是说，作用域扩展属性可以定义一个变量和常量的集合，这个集合可以作为一个单位操作。这里提供两个相当广泛的单位系统：CGSUnitBase 和 ElectronicUnitBase。当然，这些只是作为示例，毫无疑问它们可以被显著提高和扩展。

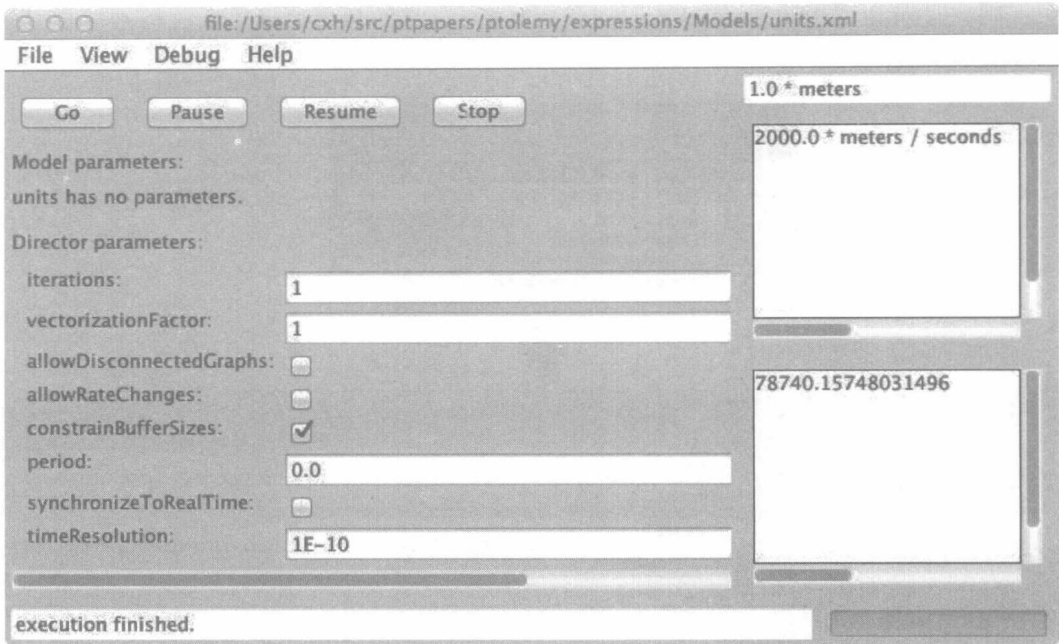


图 13-12 图 13-10 中模型的运行结果



图 13-13 单位不匹配造成异常的示例

13.8 函数表

表 13-4 三角函数

函数名	参数类型	返回类型	描述
acos	取值为 $[-1.0, 1.0]$ 的 double 或 complex	取值为 $[0.0, \pi]$ 的 double 或如果超出范围为 NaN, 或 complex	反余弦函数 complex 时: $a \cos(z) = -i \log\left(z + i\sqrt{1-z^2}\right)$
asin	范围为 $[-1.0, 1.0]$ 的 double 或 complex	范围为 $[-\pi/2, \pi/2]$ 的 double 或如果超出范围为 NaN, 或 complex	反正弦函数 complex 时: $a \sin(z) = -i \log\left(iz + i\sqrt{1-z^2}\right)$
atan	double 或 complex	范围为 $[-\pi/2, \pi/2]$ 的 double 或 complex	反正切函数 complex 时: $a \tan(z) = -\frac{i}{2} \log\left(\frac{i-z}{i+z}\right)$
atan2	double, double	范围为 $[-\pi, \pi]$ 的 double	矢量角 (注意: 参数为 (y, x) , 而不是 (x, y))。
acosh	大于 1 的 double 或 complex	double 或 complex	双曲反余弦函数 double 和 complex 时: $a \cosh(z) = \log\left(z + \sqrt{z^2-1}\right)$

(续)

函数名	参数类型	返回类型	描述
asinh	double 或 complex	double 或 complex	双曲反正弦函数 complex 时: $a \sinh(z) = \log\left(z + \sqrt{z^2 + 1}\right)$
cos	double 或 complex	范围为 $[-1, 1]$ 的 double, 或 complex	余弦函数 complex 时: $\cos(z) = \frac{\exp(iz) + \exp(-iz)}{2}$
cosh	double 或 complex	double 或 complex	双曲余弦函数 double 和 complex 时: $\cosh(z) = \frac{\exp(iz) + \exp(-z)}{2}$
sin	double 或 complex	double 或 complex	正弦函数 complex 时: $\sin(z) = \frac{\exp(iz) - \exp(-iz)}{2i}$
sinh	double 或 complex	double 或 complex	双曲正弦函数 double 和 complex 时: $\sinh(z) = \frac{\exp(z) - \exp(-z)}{2}$
tan	double 或 complex	double 或 complex	正切函数 double 和 complex 时: $\tan(z) = \frac{\sin(z)}{\cos(z)}$
tanh	double 或 complex	double 或 complex	双曲正切函数 double 和 complex 时: $\tanh(z) = \frac{\sinh(z)}{\cosh(z)}$

表 13-5 基本数学函数，第一部分

函数名	参数类型	返回类型	描述
abs	double 或 complex	double 或 int 或 long (complex 时返回 double)	求绝对值 complex 时: $abs(a + ib) = z = \sqrt{a^2 + b^2}$
angle	complex	范围为 $[-\pi, \pi]$ 的 double	角度或 complex 的参数: $\angle z$
ceil	double 或 float	double	向上取整函数, 该函数返回最小的 (接近于负无穷大) double 值, 这个值不小于参数, 且是一个整数
compare	double, double	int	比较两个数, 返回 -1、0 或 1, 分别代表第一个数小于、等于或大于第二个数
conjugate	complex	complex	复数共轭
exp	double 或 complex	范围为 $[0.0, \infty]$ 的 double 或 complex	指数函数 (e^{argument}) complex 时: $e^{a+ib} = e^a(\cos(b)+isin(b))$
floor	double	double	向下取整函数, 该函数返回最大的 (接近于正无穷大) double 值, 这个值不大于参数, 且是一个整数
gaussian	double, double 或 double, double, int, 或 double, double, int, int	double 或 arrayType(double) 或 [double]	一个或多个具有指定均值和标准误差的高斯随机变量 (见 13.4.8 节)
imag	complex	double	虚部
isInfinite	double	boolean	如果参数是无穷大则返回 true
isNaN	double	boolean	如果参数不是数字则返回 true

(续)

函数名	参数类型	返回类型	描述
log	double 或 complex	double 或 complex	自然对数 complex 时 $\log(z) = \log(\text{abs}(z) + i \angle z)$
log10	double	double	以 10 为底的对数
log2	double	double	以 2 为底的对数
max	double, double 或 double	一个与参数类型相同的标量	最大数
min	double, double 或 double	一个与参数类型相同的标量	最小数
pow	double, double 或 complex, complex	double 或 complex	第一个参数是第二个参数的幂

表 13-6 基本数学函数，第二部分

函数名	参数类型	返回类型	描述
random	无参数 或 int 或 int, int	double 或 double 或 [double]	0.0 ~ 1.0 之间的一个或多个随机数（见 13.4.8 节）
real	complex	double	实数部分
remainder	double, double	double	根据 IEEE 754 浮点数标准，求余除法（见 13.4.8 节）
round	double	long	转换成最接近的 long，在范围内选择最大的整数，如果超出范围则显示异常。如果没有参数，则返回 0。如果参数超出范围，则返回值根据符号显示为 Maxlong 或 Minlong
roundToInt	double	int	转换成最接近的 int，在范围内选择最大的整数，如果超出范围则显示异常。如果没有参数，则返回 0。如果参数超出范围，则返回值根据符号显示为 Maxint 或 Minint
sgn	double	int	如果参数为负数返回 -1，否则返回 1
sqrt	double 或 complex	double 或 complex	平方根。如果参数是小于 0 的 double 数，那么返回 NaN complex 时 $\text{sqrt}(z) = \sqrt{ z } \left(\cos\left(\frac{\angle z}{2}\right) + i \sin\left(\frac{\angle z}{2}\right) \right)$
toDegrees	double	double	弧度转化为度
toRadians	double	double	度转化为弧度
within	type, type, double	boolean	如果第一个参数与第二个参数的距离小于或等于第三个参数的值，则返回为 true。前两个参数可以是任何类型。对于复合类型、数组、记录和矩阵，如果前两个参数具有相同的结构，且每个对应的元素的距离小于或等于第三个参数，则返回为 true

表 13-7 参数和返回值类型为矩阵、数组或记录的函数，第一部分

函数名	参数类型	返回类型	描述
arrayToMatrix	arrayType(type), int, int	[type]	根据数组创建指定行列数的矩阵
concatenate	arrayType(type), arrayType(type)	arrayType(type)	连接两个数组
concatenate	arrayType(arrayType(type))	arrayType(type)	连接数组的数组
conjugateTranspose	[complex]	[complex]	返回指定矩阵的共轭转置
createSequence	type, type, int	arrayType(type)	创建一个数组，第一个元素为第一个参数，递增因子为第二个参数，数组长度为第三个参数
crop	[int], int, int, int, int 或 [double], int, int, int, int 或 [complex], int, int, int, int 或 [long], int, int, int, int	[int] 或 [double] 或 [complex] 或 [long]	给出任意类型的矩阵，返回一个在指定行列开始的子矩阵，且该矩阵行列数为指定数目

(续)

函数名	参数类型	返回类型	描述
determinant	[double] 或 [complex]	double 或 complex	返回指定矩阵的行列式
diag	arrayType(type)	[type]	返回一个对角矩阵, 由指定数组沿对角线给定值
divideElements	[type], [type]	[type]	返回两个矩阵相除值 (逐值相除)
emptyArray	type	arrayType(type)	返回一个空数组, 其元素类型与指定令牌相匹配
emptyRecord		record	返回一个空记录
find	arrayType(type), type	arrayType(int)	返回一个指数数组, 该数组元素类型与指定令牌相匹配
find	arrayType(boolean)	arrayType(int)	返回指数数组, 该指定数组的元素值为真
hilbert	int	[double]	返回一个 Hilbert 方阵, 其中 $A_{ij} = \frac{1}{i+j+1}$, Hilbert 方阵接近于奇异方阵

表 13-8 参数和返回值类型为矩阵、数组或记录的函数, 第二部分

函数名	参数类型	返回类型	描述
identityMatrixComplex	int	[complex]	返回指定大小的单位矩阵
identityMatrixDouble	int	[double]	返回指定大小的单位矩阵
identityMatrixInt	int	[int]	返回指定大小的单位矩阵
identityMatrixLong	int	[long]	返回指定大小的单位矩阵
intersect	record, record	record	返回一个记录, 该记录包括第二参数与第一个记录中相同的字段
inverse	[double] 或 [complex]	[double] 或 [complex]	返回指定矩阵的逆矩阵, 如果它是单数则报错
matrixToArray	[type]	arrayType(type)	创建包含矩阵中值的数组
merge	record, record	record	合并两个记录, 当它们有相同的记录标签时第一个优先
multiplyElements	[type], [type]	[type]	两个指定元素类型的矩阵相乘
orthonormalizeColumns	[double] 或 [complex]	[double] 或 [complex]	返回一个含正交列的相似矩阵
orthonormalizeRows	[double] 或 [complex]	[double] 或 [complex]	返回一个含正交行的相似矩阵
repeat	int, type	arrayType(type)	通过重复执行指定次数的令牌来创建一个数组
sort	arrayType(string) 或 arrayType(realScalar)	arrayType(string) 或 arrayType(realScalar)	返回一个按升序排序的数组。realScalar 是任意的标量令牌, 除了 complex 类型外
sortAscending	arrayType(string) 或 arrayType(realScalar)	arrayType(string) 或 arrayType(realScalar)	返回一个按升序排序的数组。realScalar 是任意的标量令牌, 除了 complex 类型外

表 13-9 参数和返回值类型为矩阵、数组或记录的函数, 第三部分

函数名	参数类型	返回类型	描述
sortDescending	arrayType(string) 或 arrayType(realScalar)	arrayType(string) 或 arrayType(realScalar)	返回一个按降序排序的数组。realScalar 是任意的标量令牌, 除了 complex 类型外
subarray	arrayType(type), int, int	arrayType(type)	从指定位置提取固定长度的子数组

(续)

函数名	参数类型	返回类型	描述
sum	arrayType(type) 或 [type]	type	指定数组或矩阵的元素和。如果元素不支持加法或数组为空（空矩阵将返回空值）则报错
trace	[type]	type	返回指定矩阵的迹（trace）
transpose	[type]	[type]	返回指定矩阵的转置
update	int, arrayType(type)	arrayType(type)	更改数组元素
zeroMatrixComplex	int, int	[complex]	返回一个指定行列数的零值矩阵
zeroMatrixDouble	int, int	[double]	返回一个指定行列数的零值矩阵
zeroMatrixInt	int, int	[int]	返回一个指定行列数的零值矩阵
zeroMatrixLong	int, int	[long]	返回一个指定行列数的零值矩阵

表 13-10 实用表达式评估函数

函数名	参数类型	返回类型	描述
eval	string	任何类型	计算指定的表达式（见 13.4.8 节）。
parseInt	string 或 string, int	int	从字符串中读取一个 int，如果提供了第二个参数则使用给定的数
parseLong	string 或 string, int	int	从字符串中读取一个 long，如果提供了第二个参数则使用给定的数
toBinaryString	int 或 long	string	返回参数的二进制表示
toOctalString	int 或 long	string	返回参数的八进制表示
toString	double 或 int 或 int, int 或 long	string	返回参数的字符串表示形式，如果提供了第二个参数则使用给定的数
traceEvaluation	string	string	计算指定表达式，并给出计算详情（见 13.4.8 节）

表 13-11 信号处理常用函数，第一部分

函数名	参数类型	返回类型	描述
close	double, double	boolean	如果第一个参数接近于第二（使用 EPSILON，EPSILON 是这个类的一个静态公共变量），则返回 true
convolve	arrayType(double), arrayType(double) 或 arrayType(complex), arrayType(complex)	arrayType(double) 或 arrayType(complex)	返回两个数组的卷积（是一个数组，其长度为两个参数长度之和减 1）。两个数组的卷积与多项式乘法相同
DCT	arrayType(double) 或 arrayType(double), int 或 arrayType(double), int, int	arrayType(double)	返回指定数组的离散余弦变换，使用指定（可选）长度和标准化规则（详见 13.4.8 节）
downsample	arrayType(double), int 或 arrayType(double), int, int	arrayType(double)	返回一个包括参数数组中 n 个元素的新数组，其中 n 是第二参数。如果给出了第三个参数，那么这个参数必须在 $0 \sim n-1$ 之间，它是一个数组偏移量（指出第一个输出的位置）
FFT	arrayType(double) 或 arrayType(complex) 或 arrayType(double), int arrayType(complex), int	arrayType(complex)	返回指定数组的快速傅里叶变换。如果第二个参数为 n ，那么变换长度为 2^n 。否则，该长度为大于或等于输入数组长度的下一个量。如果输入长度不符合这个长度，那么不足位补 0

(续)

函数名	参数类型	返回类型	描述
generateBartlettWindow	int	arrayType(double)	指定长度的 Bartlett (矩形) 窗口, 端点值为 0.0。如果长度为奇数, 则中心点值为 1.0。长度为 $M+1$ 时, 计算公式为 $w(n)=\begin{cases}2\frac{n}{M} & 0\leq n\leq \frac{M}{2}\\2-\frac{n}{M} & \frac{M}{2}\leq n\leq M\end{cases}$

表 13-12 信号处理常用函数, 第二部分

函数名	参数类型	返回类型	描述
generateBlackmanWindow	int	arrayType(double)	返回具有指定长度的 Blackman 窗口。长度为 $M+1$ 时, 计算公式为 $w(n)=0.42+0.5\cos\left(\frac{2\pi}{M}\right)+0.08\cos\left(\frac{4\pi n}{M}\right)$
generateBlackmanHarrisWindow	int	arrayType(double)	返回具有指定长度的 Blackman-Harris 窗口。长度为 $M+1$ 时, 计算公式为 $w(n)=0.35875+0.48829\cos\left(\frac{2\pi n}{M}\right)+0.14128\cos\left(\frac{4\pi n}{M}\right)+0.01168\cos\left(\frac{6\pi n}{M}\right)$
generateGaussianCurve	arrayType(double), arrayType(double), int	arrayType(double)	返回指定的标准偏差、范围和长度的高斯曲线。长度为标准偏差的倍数。例如, 为了获得 100 个样本的标准偏差为 1.0 ~ 4 的高斯曲线, 使用函数 generateGaussianCurve (1.0, 4.0, 100) 得到
generateHammingWindow	int	arrayType(double)	返回具有指定长度的 Hamming 窗口。长度为 $M+1$ 时, 计算公式为 $w(n)=0.54-0.46\cos\left(\frac{2\pi n}{M}\right)$
generateHanningWindow	int	arrayType(double)	返回具有指定长度的 Hanning 窗口。长度为 $M+1$ 时, 计算公式为 $w(n)=0.5-0.5\cos\left(\frac{2\pi n}{M}\right)$
generatePolynomialCurve	arrayType(double), double, double, int	arrayType(double)	返回一个多项式指定的曲线样本。第一个参数是一个多项式系数数组, 以常数项、线性项、平方项等开始。第二个参数是多项式变量开始的值。第三个参数是每个连续样本的变量增量。最后一个参数是返回数组的长度

表 13-13 信号处理常用函数, 第三部分

函数名	参数类型	返回类型	描述
generateRaisedCosinePulse	double, double, int	arrayType(double)	返回一个包含对称升余弦脉冲的数组。此脉冲被广泛用于通信系统中, 并且被称为“升余弦脉冲”(也称为升余弦滤波器), 因为它傅里叶变换得到的取值范围是从类似矩形(如超过的带宽频率范围是零空值的情况)到升余弦中升到非负的部分(对于超过的带宽频率范围 1.0 的情况)。返回数组的元素是下列函数的样本:

(续)

函数名	参数类型	返回类型	描述
generateRaisedCosinePulse	double, double, int	arrayType(double)	$h(t) = \frac{\sin\left(\frac{\pi t}{T}\right)}{\frac{\pi t}{T}} \times \frac{\cos\left(\frac{x\pi t}{T}\right)}{1 - \left(\frac{2\pi t}{T}\right)^2}$ <p>其中 x 是多余的带宽 (第一个参数), T 是从脉冲中心到第一过零点 (第二个参数) 的样本的数目。样本的采样间隔为 1.0, 返回数组是对称的, 长度为第三个参数。如果多余的带宽为 0.0, 则该脉冲是正弦脉冲</p>
generateRectangularWindow	int	arrayType(double)	返回元素为 1.0 的指定长度的数组。这是一个长方形的窗口
IDCT	arrayType(double) 或 arrayType(double), int 或 arrayType(double), int, int	arrayType(double)	返回指定数组的反离散余弦变换, 使用指定的 (可选) 长度和标准化规则 (请见 13.4.8 节)
IFFT	arrayType(double) 或 arrayType(complex) 或 arrayType(double), int arrayType(complex), int	arrayType(complex)	指定数组的快速傅里叶逆变换。如果第二个参数的值为 n , 那么变换的长度为 2^n 。否则, 该长度为大于或等于输入数组长度的下一个幂。如果输入长度不符合这个长度, 那么不足位补 0

表 13-14 信号处理常用函数, 第四部分

函数名	参数类型	返回类型	描述
nextPowerOfTwo	double	int	返回两个大于或等于参数的下一个幂
poleZeroToFrequency	arrayType(complex), arrayType(complex), complex, int	arrayType(complex)	给出极点位置的数组、零点位置的数组、一个增益项和一个数组大小, 返回表示由这些极点、零点和增益所指定的代表频率响应的指定大小的数组。这是通过围绕单位圆周计算得到零点的距离, 并由得到的到 poles 距离来进行划分的, 最后乘以增益
sinc	double	double	返回 sinc 函数 $\sin(x)/x$, 特别要注意, 如果参数是 0.0, 则返回 1.0
toDecibels	double	double	返回 $20 \times \log_{10}(z)$, 其中 z 是参数
unwrap	arrayType(double)	arrayType(double)	修改指定的数组来打开角度。也就是说, 如果连续值之间的差大于 π , 那么第二个值被修改 2π 的倍数, 直到差小于或等于 π 。此外, 修改第一个元素以便它与零的差小于或等于 π
upsample	arrayType(double), int	arrayType(double)	返回一个新的数组, 它是在输入数组中的每个连续样本之间插入 $n-1$ 个 0 的结果, 此处 n 是第二个参数。返回的数组长度为 nL , 其中 L 是参数数组的长度, $n > 0$

表 13-15 其他函数, 第一部分

函数名	参数类型	返回类型	描述
asURL	string	string	返回参数的 URL 表示
cast	type1, type2	type1	返回由第二个参数转换的与第一个参数类型一样的结果, 如果转换无效, 则返回异常

(续)

函数名	参数类型	返回类型	描述
constants	none	record	返回一个记录, 它识别表达语言中的所有全局定义常量
findFile	string	string	给定一个与用户目录、当前目录或其他路径有关的文件名, 返回第一个匹配的正确文件名, 如果没有找到匹配的名称则返回原文件名
filter	function, arrayType(type)	arrayType(type)	提取一个子数组, 该数组的所有元素的给定的判定函数返回值为真
filter	function, arrayType(type), int	arrayType(type)	提取一个有限大小的子数组, 该数组的所有元素的给定的判定函数返回值为真
freeMemory	none	long	返回可用的内存大小的近似字节数
iterate	function, int, type	arrayType(type)	返回一个数组, 它是第三个参数对应的指定函数运行的结果, 将其应用到应用程序的结果中, 再重复运算得到一个数组, 数组长度为第二个参数的值
map	function, arrayType(type)	arrayType(type)	返回一个数组, 它是以运行指定应用程序的结果作为参数送入指定函数所得到的数组元素。
property	string	string	返回具有指定名称的系统属性, 如果没有则返回空。有些有用系统属性包括 java.version、ptolemy.ptII.dir、ptolemy.ptII.dirAsURL 和 user.dir

表 13-16 其他函数, 第二部分

函数名	参数类型	返回类型	描述
readFile	string	string	得到指定文件中的字符串文本, 如果找不到文件, 则返回异常。文件可以在当前工作目录 (user.dir)、用户的主目录 (user.home) 或类路径中。readfile 函数通常与 eval 函数一起使用
readResource	string	string	得到指定资源中的字符串文本 (可以在相关类路径中找到), 无法找到该文件时返回异常
totalMemory	none	long	返回当前对象所使用的字节数与将要使用的分配字节数的和
yesNoQuestion	string	boolean	查询用户确定或否定, 并返回一个布尔值。如果 GUI 可用, 那么这个函数打开一个对话框, 否则将使用标准输入和输出

类型系统

Edward A. Lee、Marten Lohstroh 和 Yuhong Xiong

在程序语言中，**类型系统**（type system）为每个变量指定类型。从逻辑上来讲，**类型**（type）就是变量可以表示值的范围的集合。例如，根据 IEEE 754 浮点算数标准，双精度单点型是由所有 64 位表示的双精度浮点数的集合。**强类型系统**（strong type system）可以防止程序员使用 64 位变量定义一个指向内存的指针变量或长整型（long）变量（Liskov and Zilles, 1974）。Java 有一个强类型系统，但 C 没有。Cardelli and Wegner（1985）给出了一个很好的类型系统介绍。

Ptolemy II 的类型系统将类型与模型的每个端口和参数联系起来。Ptolemy II 使用**类型推断**（type inference）机制，即根据参数和端口的用途来推断它们的类型。通常，模型生成器都不需要声明其类型。

在程序语言中，在编译时检查的类型称为**静态类型**（statically typed）。在 Ptolemy II 中，类型检查仅在一个模型执行之前进行，即只在预初始化和初始化阶段之间进行类型检查。类型检查一旦执行，就认为 Ptolemy II 是静态类型。

尽管类型系统很健壮（例如，端口不可能接收到一个与它声明的类型不相符的令牌），但是它还是有漏洞的。尤其用户可以用 Java 定义他们自己的角色，却没有很好地对这些角色进行规范。例如，一个角色可能已经声明了一个整型（int）输出端口，却尝试通过这个端口输出一个**字符串类型**（string）。为了能发现这样的错误，Ptolemy II 可在运行时执行类型检查。尽管这样的错误很少（除非用户定义自己的参数），但创建一个在运行过程中避免类型错误的模型是可行的。这些错误并不能被静态类型检查发现。

幸运的是，得益于静态类型检查，当令牌从端口输出时，运行时可以自动执行类型检查。运行时类型检查器只简单地比较产生的令牌类型与输出端口（静态）类型是否相符。按这种方式，就能尽早发现类型错误（可能是在令牌产生而不是在其使用时）。然而，这并不能保证一个角色收到的令牌类型与运算是兼容的。因此，如果没有正确编写角色，那么角色本身仍将抛出一个运行时类型错误。

Ptolemy II 的类型系统支持**多态性**（polymorphism），这里角色可以对不同数据类型进行运算。为了方便构建多态角色，类型系统提供了一种称为**自动类型转换**的机制，在数据类型转换不会丢失信息的假设情况下，这种机制允许一个组件接收多种数据类型（通过自动将这些数据类型转换成同一种数据类型）。多态性极大地提高了静态类型中角色的可重用性，特别是在结合了**类型推断**机制时。本章将展示这些机制如何集成在 Ptolemy II 静态类型检查框架中，并将着重介绍这些机制如何有助于创建正确的模型。

14.1 类型推断、转换和冲突

在 Ptolemy II 模型中，端口的类型是从模型中推断出来的，它受到角色和基础设施的约

束。本书将解释这些约束如何产生以及如何推断，首先给出一个直观的解释。

例 14.1 图 14-1 中的模型计算并显示了 $1-\pi$ （这是一个呆板的模型，因为整个模型可以用表达式 $1-\pi$ 代替，但它对类型系统进行描绘）。该模型包含了一个称为 ShowTypes（类型显示）的属性，该属性可以在 `utilities`→`Analysis` 中找到。这个属性可以使 Vergil 在每个端口旁边显示其类型（如果端口有名字，则显示，否则不显示端口名）。如图 14-1 所示，最初每个端口的类型都是未知的（unknown）。

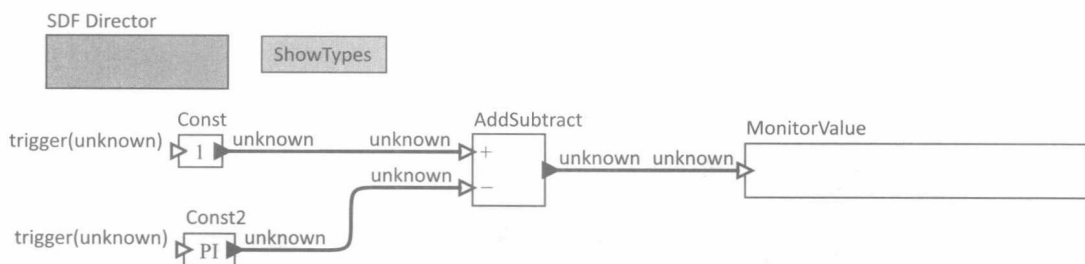


图 14-1 说明类型推断与转换的简单示例

在执行的第一阶段，即预初始在（preinitialize）阶段，进行端口类型的，在此之后更新显示，如图 14-2 所示。Const 角色的输出端口的类型是由其（value）参数决定的，分别为 1 和 π 。如果将一个参数改为 $1+i$ ，那么输出类型将变成 complex。如果将它改为 {1, 2}，那么输出类型将变成一个具有两个整型元素的数组类型 `arrayType(int, 2)`。

注意在图 14-2 中，AddSubtract 角色的输入端口有两 double 类型。AddSubtract 角色规定它的两个输入端口必须具有相同的类型。当 AddSubtract 角色从 Const 角色接收到一个 int 令牌时，输入端口将自动把该令牌转换为 double 类型。本章将详细解释哪种转换是允许发生的，直观来说，如果不发生信息丢失则允许转换。

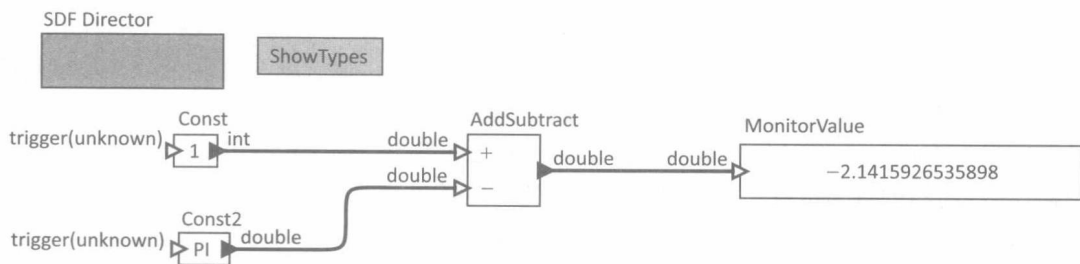


图 14-2 执行后

注意在图 14-2 中，AddSubtract 角色的输出端口和 MonitorValue 角色的输入端口都是 double 类型。MonitorValue 角色可以接受任意的输入类型，因为它只是简单地显示令牌的字符串表示，且每一个令牌都有一个字符串表示。

前面的例子说明在模型的一个部件中（在一个模型可以产生深远影响的一部分）的参数类型可能会对模型的其他部件产生很大影响。类型系统确保其一致性。在构建模型时出现类型错误是很正常的，接下来的例子给出了说明。

例 14.2 如图 14-3 所示。在这个例子中，将 MonitorValue 角色换成了 SequencePlotte 角色，并让 Const 角色产生一个 complex 值。类型推断确定 AddSubtract 角色的输出是复数。但是 SequencePlotter 角色需要一个 double 类型的输入。一个复数类型令牌不能无损地转换

为 double 类型令牌，所以在执行模型时将得到以下异常信息：

```
ptolemy.actor.TypeConflictException: Type conflicts occurred on
the following inequalities:
(port .TypeConflict.AddSubtract.output: complex) <=
(port .TypeConflict.SequencePlotter.input: double)
in .TypeConflict
```

这个出错信息提示模型中有一个类型约束（type constraint）条件没有满足。该类型约束是：AddSubtract 角色的输出端口的类型必须小于或等于（ \leq ）SequencePlotter 角色的输入端口的类型。而且，输出端口的类型是 complex，而输入端口的类型是 double。出错端口和它们的容器在图中高亮显示。

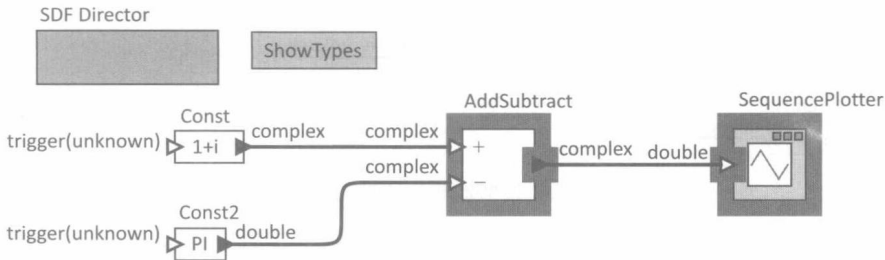


图 14-3 类型冲突

在上面的例子中，类型约束是以不等式的形式给出的，声称一个端口的类型必须“小于或等于”另一个端口的类型。这是什么意思呢？

直观地说，如果一种类型可以无损地转换为另一种类型，那么就认为前一种类型小于后一种类型。接下来将分析这种不等式的关系。

14.1.1 自动类型转换

图 14-4 显示了允许的自动类型转换，它描述了 Ptolemy II 中的类型格（type lattice）。在图 14-4 中，如果从第一个类型到第二个类型有上升路径，那么就允许从第一个类型转换到第二个类型。这种关系指的是类型上的偏序（partial order）（见第 14 章补充阅读：格是什么），所以可以说，如果第一个类型小于或等于第二个类型，那么就允许转换。这种称为格的偏序具有简洁的数学结构，它能提供高效的类型推断和类型检查。

自动类型转换发生在一个角色从其输入端口取回数据时^①。端口类型由前期执行决定，运行时类型检查保证通过输出端口发送的令牌与下游输入端口类型兼容（即类型格允许这样的转换）。这是由于类型约束是两个端口之间的连接施加的。在 14.1.2 节中将解释这些类型约束。如果令牌不兼容，那么运行时类型检查器将在从发送令牌之前抛出异常。因此，运行时类型错误应尽早检测出来。

在表达式语言中，可以使用 cast 函数来实现强制的类型转换，它是多内置函数中的一个。函数 cast(newType, value) 能将指定值转换为指定类型。请参见 13.4.3 节和表 13-15 中关于 cast 函数的信息。

① 有些角色在它们的输入端口上禁用自动类型转换，因为它们不需要转换。例如，AddSubtract 角色和 Display 角色接受任何类型的令牌，因为它们使用的方法能继承所有令牌类型。这些角色通过调用指令 portName.setAutomaticTypeConversion(false) 来禁用自动转换。

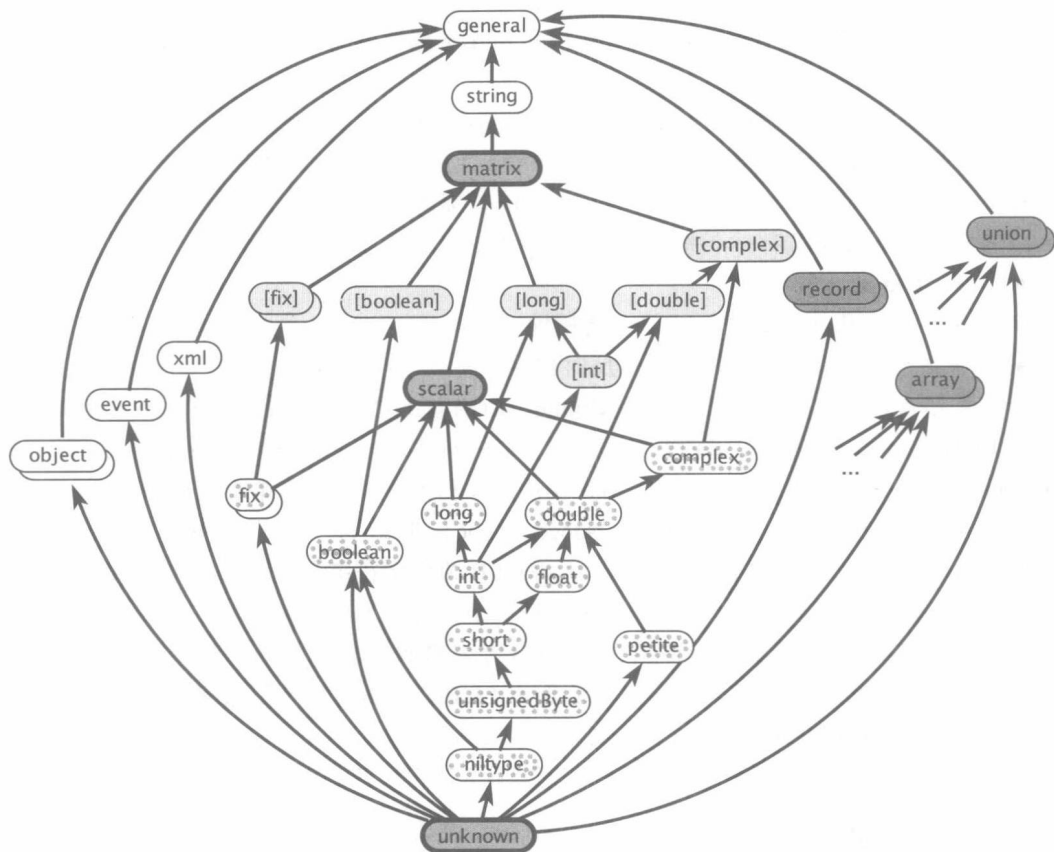


图 14-4 类型格 (type lattice)。图中深灰色填充边框加粗的为不能实例化的类型。浅灰色点填充的为标量 (scalar) 类型，浅灰色填充的是矩阵类型 (matrix)，深灰色填充的是复合类型 (composite)。双重视窗显示的复合类型和固定点类型 (fix) 属于无限子格

类型格是基于无损转换原则的基础上构造的。只要数据令牌的重要信息不丢失，就自动允许转换。这样的转换类似于 Java 中的扩大转换 (widening conversion)。例如，将一个 32 位带符号整数转换为 64 位 IEEE 标准双精度浮点数是允许的，因为每个整数可以按一个浮点数来精确表示。另一方面，丢失信息的数据类型转换将不会自动进行。

补充阅读：格 (Lattice) 是什么

格是一种数学结构，它能有效解决类型约束问题。格是与 CPO 有关的带有特定次序关系的集合 (见第 5 章补充阅读：cpo、连续函数和固定点)。详情请参考文献 Davey and Priestly (2002)。

首先，全序 (total order) 是指一个有序集合 S ，用 \leq 表示，其中集合中任意两个元素都是有序的。特别地，对于任意 $x, y \in S$ ，都有 $x \leq y$ 或 $y \leq x$ (或者两者都满足，也就是 $x = y$)。例如，假设集合 S 是一个整数集合， \leq 表示普通算术顺序，那么 (S, \leq) 就是一个全序。

偏序限制没有那么严格，规定集合中任意两个元素有顺序即可。例如，偏序 (S, \leq) ，其中 S 是一个集合的子集， \leq 是子集之间的关系，通常用 \subseteq 表示。特别地，如果

A, B 是 S 中的集合, 那么 $A \subseteq B$ 不成立, $B \subseteq A$ 也不成立。例如, 令 $A = \{1, 2\}, B = \{2, 3\}$, 那么 A, B 都不是对方的子集。

另一种偏序关系是字符串的前缀顺序 (prefix order)。例如, S 为字母数字字符序列的集合, 那么对于两个字符串 $x, y \in S$, 如果 x 是 y 的前缀, 那么就认为 $x \leq y$ 。例如, 如果 $x = abc, y = abcd$, 那么 $x \leq y$ 。如果 $z = bc$, 那么 $x \leq z$ 和 $z \leq x$ 都不成立。形式化地, 偏序是一个集合 S 和一个关系 \leq , 且对于所有 $x, y, z \in S$, 都有

- $x \leq x$ (自反性)。
- 如果 $x \leq y$ 且 $y \leq z$, 则 $x \leq z$ (传递性)。
- 如果 $x \leq y$ 且 $y \leq x$, 则 $x = y$ (反对称性)。

如果存在最小上界 (Least Upper Bound, LUB) 且它是偏序 (S, \leq) 的子集 $U \subseteq S$, 那么 LUB 是最小元素 $x \in S$ 使得对于每个 $u \in U$ 有 $u \leq x$ 。如果存在 U 的最大下界 (Greatest Lower Bound, GLB), 那么 GLB 是最大元素 $x \in S$, 使得每个 $u \in U$ 有 $x \leq u$ 。例如, 在前缀顺序中, 如果 $x = abc, y = abcd, z = bc$, 那么 $\{x, y\}$ 的 LUB 为 y 。 $\{x, z\}$ 的 LUB 不存在。 $\{x, y\}$ 的 GLB 为 x 。 $\{x, z\}$ 的 GLB 为空字符串, 它是所有字符串的前缀。

格为一种偏序关系 (S, \leq), 其中 S 的每个子集都有 GLB 和 LUB。子集顺序是一个格, 因为 LUB 可以通过集合并得到, 而 GLB 可以根据集合交得到。但是, 字符串的前缀顺序不是一个格, 因为两个字符串可能没有 LUB。前缀顺序是下级半格 (lower semi-lattice), 因为字符串集合的 GLB 总是存在。

14.1.2 类型约束

模型施加一些约束来用于类型推断。约束可以理解为两个端口类型之间的不平等关系。它要求一个端口的类型小于或等于 (可无损转换成) 另一个端口的类型。

在 Ptolemy II 技术中, 类型兼容规则要求输出端口的类型小于或等于与它所连接的所有输入端口的类型,

$$\text{outType} \leq \text{inType} \quad (14-1)$$

这个约束保证在数据转换时允许自动转换发生。输出端口与输入端口之间的每个连接都建立一个类型约束来执行该规则。

例 14.3 如图 14-2 所示, Const 角色产生一个 int 类型, 而 AddSubtract 角色接收一个 double 类型。在图 14-4 中, 可以看到 int 比 double 小, 所以是满足类型兼容规则的。

除了角色之间连接施加的约束外, 大多数角色也会施加约束。例如, AddSubtract 角色声明它的输出类型必须大于或等于其输入类型, 且两个输入端口的类型是相同的。该等式约束相当于两个不等式约束, 如下所示:

$$\begin{aligned} \text{plus} &\leq \text{minus} \\ \text{minus} &\leq \text{plus} \end{aligned}$$

这里, plus 和 minus 是 AddSubtract 角色的输入端口。

许多角色施加一个默认类型约束 (default type constraint), 它要求无约束的输入小于或等于每个无约束的输出。默认情况下, 对于每个没有显式类型约束的输入端口和输出端口集合, 角色都隐式地包含这个类型约束。

例 14.4 有些角色对令牌进行操作而忽视了令牌的实际类型。例如, DownSample 角色

就不在乎通过它的令牌类型，所以它并不显式地声明任何类型约束。默认的类型约束使类型信息通过这个角色从输入向输出传播。

默认情况下，对于不声明输入端口的角色，它们输入端口的类型取决于上游模型（除非该模型可进行反向类型推断 (backward type inference)，将在 14.1.4 节详细说明这类情况）。

例 14.5 MonitorValue 角色显示它接收到的令牌值，且它接受任意类型的输入。默认情况下，它不声明输入，这样无论上游提供什么类型都能解决。例如，在图 14-5 中，将 SequencePlotter 角色换为 MonitorValue 角色，就能解决图 14-3 中的类型冲突。已被解决的输入类型 complex 由上游角色决定。

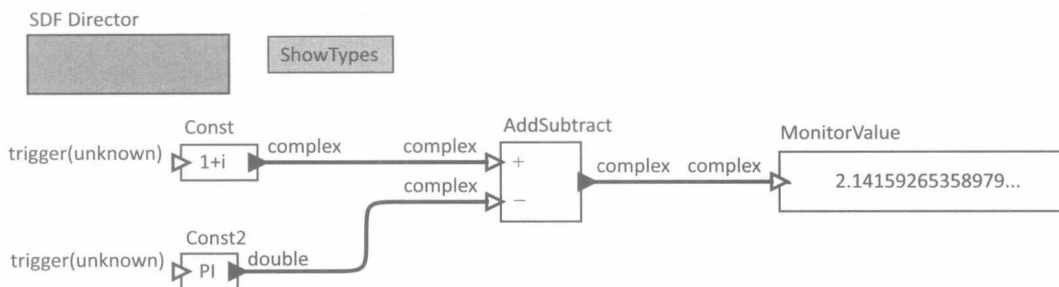


图 14-5 通过使用一个可接受任意输入类型的 MonitorValue 角色来解决图 14-3 的类型冲突问题

14.1.3 类型声明

有时，在模型中来自数据源的信息不足以推断出数据类型。

例 14.6 考虑图 14-6 中的模型。该模型的目的是评估用户在 shell 中输入的表达式，但类型解析失败。ExpressionToToken 角色接收一个输入字符串，将其用 Ptolemy 表达式语言来表示（参见第 13 章），并计算该表达式。计算的结果在输出端口产生。因为没有办法预测用户在 shell 输入的类型，所以没有足够的信息来推断类型。ExpressionToToken 的输出端口类型仍然是未知的 (unknown)。

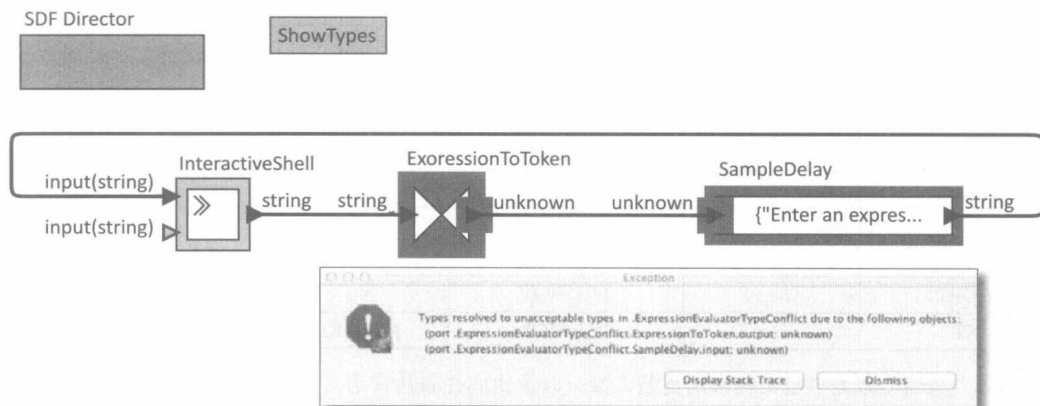


图 14-6 一个评估用户输入表达式的模型，但是没有足够的信息从数据源推断出类型

以上问题都可以通过反向类型推断 (backward type inference) (将在后面讨论) 或通过明确声明端口的类型的方式来解决，如下例所示。

例 14.7 如图 14-7 所示，可以强制规定 ExpressionToToken 角色的输出类型是通用数

据类型 (general data type)。下游类型也解析为通用型 (general)。

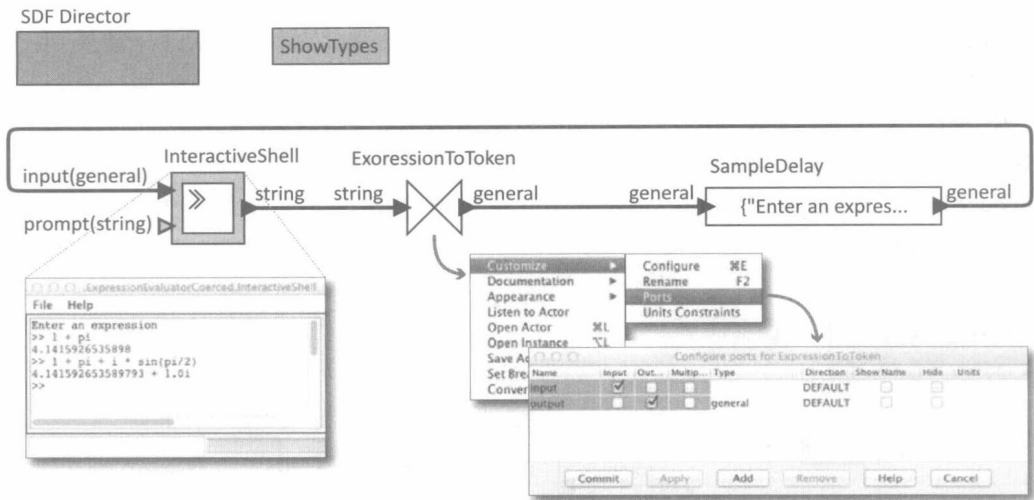


图 14-7 通过在端口配置对话框的类型到输入类型来声明端口的类型

在端口对话框的类型列中，可以输入表达式语言中的任何表达式。无论表达式是哪种类型，都成为相应端口的声明类型。简而言之，Ptolemy II 提供了一些指定类型的内置变量。例如，`general` 就是一个通用数据类型令牌。类似地，`double` 是一个 `double` 类型的令牌（碰巧是 0.0，值不重要）。表 14-8 列出了内置变量名及其对应的类型。

基础类型段	表达式	描述	基础类型段	表达式	描述
UNKNOWN	unknown	数据类型格的末端元素	INT_MATRIX	[int]	32 位整数矩阵
ARRAY_BOTTOM		未知数据类型数组	LONG	long	64 位整数
BOOLEAN	boolean	布尔型（真或假）	LONG_MATRIX	[long]	64 位整数矩阵
BOOLEAN_MATRIX	[boolean]	布尔型数据矩阵	OBJECT	object	对象类型
UNSIGNED_BYTE	unsignedByte	无符号 byte 型	ACTOR		角色型
COMPLEX	complex	复数	XMLTOKEN		XML 型
COMPLEX_MATRIX	[complex]	复数矩阵	SCALAR	scalar	标量数
FLOAT	float	32 位 IEEE 浮点数	MATRIX		未知数据类型矩阵
DOUBLE	double	64 位 IEEE 浮点数	STRING	string	字符串
DOUBLE_MATRIX	[double]	浮点数矩阵	GENERAL	general	任意类型
FIX	fixedpoint	定点数	EVENT		事件（空令牌）
FIX_MATRIX	[fixedpoint]	定点数矩阵	PETITE		-1 ~ 1 的双精度浮点数
SHORT	short	16 位整数	NIL	nil	空值类型
INT	int	32 位整数	RECORD		记录类型

图 14-8 基础类型类中定义的常用类型常量，以及他们在表达式语言中对应的名称（如果存在的话）

14.1.4 反向类型推断

目前为止讨论的所有例子中，模型的类型推断都是正向进行的，常数或固定输出类型都是承接类型，都是由上游类型的常数或固定输出类型解析出来而得出的。也就是说，类型信

息传送都是同一个方向，即执行期间令牌发送的方向。类型兼容规则（14.1 节）对输出端口没有施加有用的约束，因为如果 outType 为未知的（unknown），类型格的底部元素是满足的。然而，正向类型推断（forward type inference）通常够用，因为在大多数模型中数据源能提供这些数据的具体类型信息。

然而，在图 14-6 和图 14-7 中的模型没有这个属性。首先，由于该模型形成了一个回路，所以没有明确的“数据源”。所以每个角色即是其他角色的上游也是下游。此外，ExpressionToToken 角色不能提供任何关于它输出类型的信息。

Ptolemy II 类型系统可选择反向类型推断来解决这个问题。为了进行反向类型推断，在模型顶层将 enableBackwardTypeInference 参数设为 true，如图 14-9 所示。这样就能达到 3 个效果。第一，使特定角色不需要严格限制输入端口接收的数据以便将这些端口声明为 general 类型。例如，InteractiveShell 和 Display。第二，它允许类型约束进行逆向（上游）类型推理。具体来说，它给类型兼容规则（14.1 节）增加了额外的约束。这个额外的约束就是，每个输出端口的类型都必须大于或等于它所连接的所有输入端口类型的最大下界（GLB）。第三，对于没有明确限制端口类型关系的每个角色，它都施加了一个默认类型约束，这个默认类型约束要求它的输入端口类型大于或等于其输出端口类型的最大下界。这些额外的约束条件对于解决类似图 14-7 所示的类型强制转换等类型问题来说是足够的。

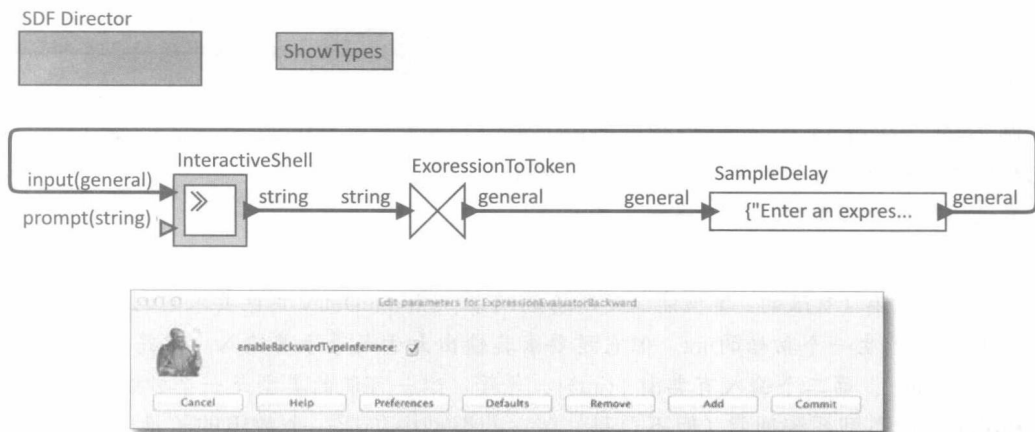


图 14-9 开启反向类型推断，允许类型约束作用于上游

问题的根源在于 ExpressionToToken 角色，它本身不能明确指出其输出类型。对它输出类型的约束必须产生于其输出令牌如何使用，而不是角色本身。在图 14-9 中，InteractiveShell 角色的输入接受 general 类型，因此它可以作用于上游 ExpressionToToken 角色的输出。当输出端口类型没有声明时，比如 ExpressionToToken 角色，那么通过反向类型推断就能发现与下游类型约束兼容的最通用的类型。

14.2 结构化类型

14.2.1 数组

结构化的类型包括那些聚集了其他任意类型的令牌，比如数组和记录。如 13.3 节所述，一个数组就是一组有序的令牌，它们具有相同的类型。记录包含一组带标签的令牌，类似一个 C 语言中的结构体。它是非常实用的对多个相关信息进行分组方式。在图 14-4 的类型格

中，记录类型与除了未知的类型（unknown）、字符串（string）和通用类型（general）以外的所有基本类型都是不可比较的。数组类型比较复杂，因为任何类型都比类型格中的数组类型数小，也就是图 14-4 中数组类型的底部断开连接的行。值得注意的是，格节点数组（array）和记录（record）实际上代表了无限种类的类型，所以类型格本身也是无限的。

对于任意类型 a ，满足以下类型关系：

$$a < \{a\}$$

一个值可以无损地转换成一个值数组。且，

$$a < b \Rightarrow \{a\} < \{b\}$$

综上所述，定义是递归的，故：

$$a < \{a\} < \{\{a\}\} < \{\{\{a\}\}\} \dots$$

例 14.8 $\text{int} \leq \text{double}$ ，可以推出：

$$\begin{aligned} \text{int} &< \{\text{int}\} \\ \{\text{int}\} &< \{\text{double}\} \\ \text{int} &< \{\text{double}\} \\ \text{int} &< \{\{\text{double}\}\} \\ &\dots \end{aligned}$$

这些类型关系的结果是任何特定数组类型有一个到类型格顶端的无限路径。这可能导致类型推断不收敛的情况。

例 14.9 在图 14-10 的模型中，Expression 角色构建包括一个元素的数组，即它的输入令牌。当你尝试运行这个模型时，发生异常，显示：

```
Large type structure detected during type resolution
```

这是因为没有（有限的）类型能满足所有的约束。SimpleDelay 角色要求其输出端口至少是 int，因为它产生一个初始的 int。但它还要求其输出大于或等于其输入。它将接收的第一个输入有类型 {int}，第二个输入有类型 {{int}}，等等。唯一可能的类型是一个数组的无限嵌套。

Ptolemy II 类型系统通常（但不总是）在它的类型中包括一个数组的长度（length）。如果要显式地查询令牌的类型，可以在表达式计算器（一个输入窗口）中输入下面的命令：

```
>> {1, 2}.getType()
object(arrayType(int,2))
```

因此，arrayType(int, 2) 是一个长度为 2 的数组类型，而 arrayType(int) 是一个长度不确定的数组类型。在类型中包括数组长度可以让类型系统检测出更多的错误。

一般来说，特定长度的数组类型与不同长度的数组类型是不可比较的，而且可以转换成任意一个未知长度（与元素类型兼容）的数组类型。标量可以转换为长度为 1 的数组类型。

有趣的是，当用表达式 {int} 指定一个数组类型时，实际上是指定了 arrayType(int, 1)，这更明确。为此，除非你想明确约束数组类型，否则最好通过 arrayType(int) 定义一个数组类型。

14.2.2 记录

两个记录类型之间的次序关系遵循标准的深度子类型化（depth subtyping）和广度子类

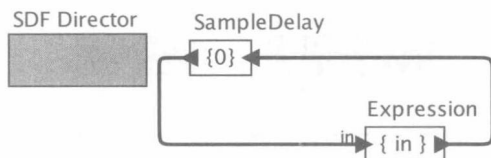


图 14-10 类型推断不收敛的例子

型化 (width subtyping) 关系 (Cardelli, 1997)。在深度子类型化中, 如果记录类型 c 的字段是记录类型 d 中对应字段的子类型, 那么记录类型 c 是记录类型 d 的一个子类型 (即, $c \leq d$)。例如,

```
{x = string, y = int} <= {x = string, y = double}
```

在广度子类型化中, 带有更多字段的记录是带有更少字段记录的子类型, 例如,

```
{x = string, y = double, z = int} <= {x = string, y = double}
```

广度子类型化的规则有点儿不合常理, 因为它在类型转换时会丢失信息或丢弃记录的额外字段。然而, 它符合 “is a” 类型关系, 即, 如果 a 是 b , 则 $a \leq b$ 。在这种情况下, 具有更多字段的记录 “是一个” 具有更少字段的记录的实例, 若反过来就不正确了。

14.2.3 联合体

另一个结构化类型就是联合体类型。它允许用户创建一个包含任意类型数据的令牌, 但每次只能是一种类型。这类似于 C 语言中的联合体。在类型系统文献中联合体类型也称为多样类型 (variant type)。联合体的广度子类型化关系与记录类型的关系相反。也就是说, 短的联合体是长的联合体的一个子类型。而且这符合 “is a” 类型关系的原则。

这种广度子类型化关系可以导致从一个特定的联合体类型到类型格的顶部有无限数量的类型。这再次表明, 类型推断可能是不收敛的。Ptolemy II 类型系统将在有限数量的步骤之后对类型推断, 并提示是否发生类型错误。

14.2.4 函数

最后一个结构化类型就是 13.4.4 节描述的函数。函数可以有多个参数并返回一个值。类型系统支持函数类型, 其中参数是已声明类型, 返回类型是已知的。在输入和输出之间逆变 (反相关) 的情况下函数类型是相关的。即, 如果 `function(x:int y:int)int` 是具有两个整型参数的一个函数, 并返回一个整数, 那么

```
function(x:int, y:int) int <= function(x:int, y:int) double
function(x:int, y:double) int <= function(x:int, y:int) int
```

这里的逆变概念适用于函数的自动类型转换。通过添加一个从 `int` 到 `double` 的返回值的转换, 返回 `int` 的函数可以转换为返回 `double` 的函数。然而输入为 `int` 的函数不能转换为输入为 `double` 的函数, 因为并没有从 `double` 到 `int` 的自动类型转换。也就是说, 如果函数在以前无法接收 `double` 型的情况下, 突然能接收 `double` 型的参数, 但实际上并无 `double` 到 `int` 型的自动转换。函数较少去考虑它们的输入类型, 而是因为函数关注更多的是自身的输出, 所以函数属于次序关系中比较低层的类型格级。

参数的名字不会影响两个函数类型之间的关系, 因为参数绑定仅取决于参数的顺序。此外, 具有不同参数个数的函数 (不同的参数数量 (arity)) 被认为是不可比较的。

可以用作任何其他令牌的函数类型, 通常称为高阶类型系统。函数令牌使用的一个例子如图 2-44 所示, 详见第 2 章补充阅读: 移动代码。

14.3 角色定义中的类型约束

12.4 节介绍了如何用 Java 编写角色。在本节中, 我们将学习如何在 Java 定义角色约束角色的类型。

在模型执行前的设置阶段，类型约束是从模型中所有对类型强加限制的实体中收集的（例如，TypedIOPort、TypedAtomicActor 或 Parameter 的实例）。角色可以通过将类型约束存储在相关端口或参数的对象实例中来设置类型约束，或者使用 TypedAtomicActor 的 customTypeConstraints 方法来设置它们。

大部分参数都有一个简单的类型约束，它确保输出的类型大于或等于参数的类型。在构造函数中输入以下语句就可以实现这个简单的类型约束：

```
portName.setTypeAtLeast(parameterName);
```

也可以如下定义：

```
protected Set<Inequality> _customTypeConstraints() {
    Set<Inequality> result = new HashSet<Inequality>();
    result.add(new Inequality(parameterName.getTypeTerm(),
        portName.getTypeTerm()));
    return result;
}
```

这称为相对类型约束（relative type constraint），因为它限制了一个对象相对于另一个对象的类型。另一种相对类型约束的形式强制要求两个对象必须有相同的类型，但不指定类型：

```
portName.setTypeSameAs(parameterName);
```

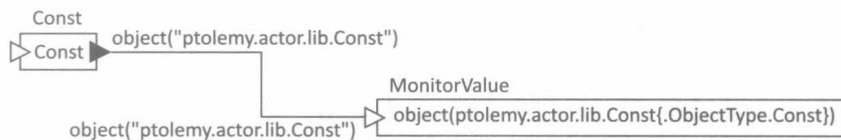
这些约束条件可以以反序指定：

```
parameterName.setTypeSameAs(portName);
```

显然，效果是一样的。

补充阅读：对象类型

在图 14-4 中，最左边的 object（对象）类型是非常强大的（因此要慎用）。object 类型的令牌代表一个 Java 对象，如 Ptolemy 角色。假设有如下一个模型：



Const 角色的值（vaule）设置为 Const，它评估 Const 角色本身。Const 角色的名称为“Const”，表达式 Const 计算该角色本身的值。因此，Const 输出端口的推断类型是 object，它的输出是角色本身。

object 类型不是单一的一种类型，而是无限多种类型，如图 14-4 中的双椭圆所显示那样。每个不同的 Java 类都有一个特定的 object 类型。以上输出端口的类型是 object("ptolemy.actor.lib.Const")，因为 Const 角色是 Java 类 ptolemy.actor.lib.Const 的一个实例。

如果 Java 类 A 是 Java 类 B 的一个子类，那么类型 object("A") 比 object("B") 小。例如，ptolemy.actor.lib.Const 实现 Java 接口 ptolemy.actor.Actor，所以

```
object("ptolemy.actor.lib.Const") <= object("ptolemy.actor.Actor")
```

在上面模型的变体中（如下图所示），MonitorValue 角色的输入端口强制转换为

```
object("ptolemy.actor.Actor").
```



一个类似的转换可以用 `cast` 函数来实现：

```
cast(object("ptolemy.actor.Actor"), Const)
```

最通用的对象类型是 `object`（毫无争议）。称为“`null`”的预定义对象令牌（`object token`）就有这种类型。

补充阅读：对象令牌的调用方法

表达式语言允许用对象令牌的 Java 类中定义某些方法（请见第 14 章补充阅读：对象类型）。例如，在一个包括角色名为 `C` 的模型中，表达式中的 `C` 就可能表示这个角色。

Java 的方法可能在对象令牌概况对象时

被整合引用在对象令牌上调用。例如，在下



面的模型中，`Const` 角色输出 `Const` 角色所包含的端口数目：

补充阅读：Petite 类和无符号字节数据类型

数据类型 `petite` 用于表示 $-1.0 \sim 1.0$ 之间的实数。它用来模拟一些诸如 DSP 处理器这样的专用处理器的行为，这些处理器有时会在上述值域内进行定点运算（fixed-point arithmetic）。`petite`（微）类型把这种数据类型近似为一个位于 $(-1.0, 1.0)$ 内的 `double` 类型。在表达式语言中，一个 `petite` 数用后缀“`p`”表示，当在 $-1.0 \sim 1$ 内运算时，得到的结果可能超出这个范围。例如，使用表达式计算可以得到：

```
>> 0.5p + 1.0p
1.0p
```

有时对于操作原始数据（raw data）（如来自网络的数据包）运算很有用的数据类型是无符号字节（unsigned byte），由如下方式指定：

```
>> lub
lub
>> -lub
255ub
```

相同的约束条件也可以表述如下：

```
protected Set<Inequality> _customTypeConstraints() {
    result.add(new Inequality(parameterName.getTypeTerm(),
        portName.getTypeTerm()));
    result.add(new Inequality(portName.getTypeTerm(),
        parameterName.getTypeTerm()));
    return result;
}
```

`_customTypeConstraints` 方法对在动态添加或删除的端口之间建立类型约束的角色非常有用，对于这些角色，把类型约束存储在各自的端口中是不安全的，因为与端口不相关联的

约束依然存在，且有可能导致类型错误。

另一种常见的类型约束是**绝对类型约束**（absolute type constraint），它固定端口的类型（例如，使端口类型成为单态而不是多态的）。定义如下：

```
portName.setTypeEquals(BaseType.DOUBLE);
```

以上函数声明了端口只能处理双精度浮点数。图 14-8 列出了定义了 `BaseType` 类中的类型常量。绝对类型约束的另一种形式是给类型设定一个上界：

```
portName.setTypeAtMost(BaseType.COMPLEX);
```

它声明可以无损地转化为 `ComplexToken` 的任何类型都是可接受的。默认情况下，对于没有声明类型约束的任何端口，自动创建的默认类型约束声明它的类型小于或等于没有声明类型约束的任何输出端口。如果存在没有约束的输入端口，但是没有缺乏约束的输出端口，那么这些输入端口将保持无约束。相反，如果存在没有约束的输出端口，但是没有缺乏约束的输入端口，那么这些输出端口将保持无约束。后者是不可接受的，除非启用了反向类型推断。可以通过覆盖 `_customTypeConstraints` 方法来禁用默认类型约束，并使它返回空。

通过下列类型约束来声明一个接受任意令牌的端口：

```
portName.setTypeAtMost(BaseType.GENERAL);
```

例 14.10 图 14-11 展示了图 12-11 中的 `Transformer` 的一个扩展。这个 `SimplerScale` 是标准 `Ptolemy II` 库中 `Scale` 角色的一个简化版。该角色每次被一个值点火时产生一个输出令牌。角色是多态的，因为它支持任何令牌类型，该令牌类型支持与 `factor` 参数相乘。在构造函数中，输出类型约束为至少应与输入和 `factor` 参数保持一致。

```

1 public class SimplerScale extends Transformer {
2     public SimplerScale(CompositeEntity container,
3         String name)
4         throws NameDuplicationException,
5             IllegalArgumentException {
6         super(container, name);
7         factor = new Parameter(this, "factor");
8         factor.setExpression("1");
9         // set the type constraints.
10        output.setTypeAtLeast(input);
11        output.setTypeAtLeast(factor);
12    }
13    public Parameter factor;
14    public Object clone(Workspace workspace)
15        throws CloneNotSupportedException {
16        SimplerScale newObject = (SimplerScale)super.
17            clone(workspace);
18        newObject.output.setTypeAtLeast(newObject.input);
19        newObject.output.setTypeAtLeast(newObject.factor);
20        return newObject;
21    }
22    public void fire() throws IllegalArgumentException {
23        if (input.hasToken(0)) {
24            Token in = input.get(0);
25            Token factorToken = factor.getToken();
26            Token result = factorToken.multiply(in);
27            output.send(0, result);
28        }
29    }
30 }
```

图 14-11 非简单类型约束的角色

注意在图 14-11 中 fire 方法如何使用 hasToken 来确保在没有输入情况下就没有输出的要求。此外,即使有多个令牌可用,每个输入通道也只使用一个令牌。这是域多态角色 (domain polymorphic actors) 的行为。另外,还要注意如何使用 Token 类的 multiply 方法。该方法是多态的。因此,这种 Scale 角色可以对任何支持乘法的令牌类型进行运算,包括所有数值类型和矩阵。

定制类型约束也有一些不好的地方,如第 12 ~ 18 行的 done 方法所示。为了能够让角色在面向角色类中正常工作,在函数构造中设置的相关类型约束必须在 clone 方法中重复。

setTypeAtLeast、setTypeAtMost、setTypeEquals 和 setTypeSameAs 方法是 Typeable 接口的一部分,Typeable 接口是由端口和参数实现的。setTypeAtMost 方法通常在输入端口上调用,以便声明 input 令牌必须满足的需求,而 setTypeAtLeast 方法通常在 output 端口上调用,以便声明输出类型的保证。_customTypeConstraints 和 _defaultTypeConstraints 方法则是基础类 TypedAtomicActor 的一部分,它们的子类可以重定义这些方法来定制它们强加的类型约束。

例 14.11 输入端口的类型不大于双精度的约束可以声明如下:

```
inputPort.setTypeAtMost(BaseType.DOUBLE);
```

注意 setTypeAtMost 和 setTypeEquals 的参数是类型,而 setTypeAtLeast 的参数是一个 Typeable 对象。这是一种常见的用法,这里 setTypeAtLeast 声明对外部提供类型的依赖,而 setTypeAtMost 和 setTypeEquals 声明对外部定义类型的约束。类型的形式是不等的,所列举的这些方法也保证了类型推断是有效的,且类型推断的结果是确定性的。

更复杂数据类型类型约束来自结构化类型,例如数组和记录。为了声明一个双精度数组的参数,可以使用:

```
parameter.setTypeEquals(new ArrayType(BaseType.DOUBLE));
```

它声明了参数或端口有一个特定的数组类型。一个更灵活的参数可能包含任意类型的数组。表达如下:

```
parameter.setTypeAtLeast(ArrayType.ARRAY_BOTTOM);
```

在一个更复杂的例子中,可能会限制一个输出端口的类型不小于被参数包含的数组元素的类型(或输入端口):

```
outputPort.setTypeAtLeast(ArrayType.arrayOf(parameter));
```

为了声明一个输出的类型大于或等于输入(或参数)的数组元素,可以如下表示:

```
outputPort.setTypeAtLeast(ArrayType.elementType(inputPort));
```

上面的代码隐含地约束输入端口是数组类型,但没有限制该数组的元素类型。上述多种约束出现在源角色中,如 DiscreteClock 和 Pulse、ArrayToSequence 和 SequenceToArray 等。检查这些角色的源代码是很有指导性的。

另一种常见的约束是,角色的输入端口接收无约束字段的记录。这个约束可以通过下面的代码来声明:

```
inputPort.setTypeAtMost(RecordType.EMPTY_RECORD);
```

假设有一个可能产生具有任意字段记录的输出端口。上面的结构是不够的，因为它没有声明类型的下界，所以在运行时就不会解决一些有用的类型。改进后的操作如下：

```
outputPort.setTypeEquals(BaseType.RECORD);
```

以下方法强制类型解决空记录。任何具有字段的记录都是空记录类型的子类型，所以它有效地声明了产生任意记录的输出。另外，反向类型推断允许从下游角色强加的类型约束来推断记录元素的类型。

为了声明一个参数可以有任意记录类型，可以这样做：

```
param.setTypeAtMost(BaseType.RECORD);
```

但是你还是需要为参数指定一个值，以便该类型可以解决一些具体问题。通过以下做法可以定义一个是空记录的默认值：

```
param.setToken(RecordToken.EMPTY_RECORD);
```

矩阵和标量这两个类型是联合体类型。这意味可以通过它们调用类型格中任何低于它们的类型可以成为联合体类型的实例。例如，一个角色参数可以声明一个输入端口类型不能大于标量 (scalar)：

```
inputPort.setTypeAtMost(BaseType.SCALAR);
```

在这种情况下，类型格中任何低于的标量的任何的类型的输入都不会立即转换，除了输入令牌类型为标量。这是非常有用的，例如，在需要对令牌进行比较的角色中（如 Limiter 角色）。该角色的 fire 方法包含以下代码：

```
if (input.hasToken(0)) {
    ScalarToken in = (ScalarToken) input.get(0);
    if ((in.isLessThan((ScalarToken) bottom.getToken()))
        .booleanValue()) {
        output.send(0, bottom.getToken());
    } else if ((in.isGreaterThan((ScalarToken) top.getToken()))
        .booleanValue()) {
        output.send(0, top.getToken());
    } else {
        output.send(0, in);
    }
}
```

这段代码依赖于声明输入端口 in 和参数 bottom 为大多数标量类型，ScalarToken 为一个基础类，代表每个标量下面的令牌类型。然后它使用 ScalarToken 类中定义的比较方法。

角色中的类型约束比这里介绍的要复杂得多。通常，源代码（和丰富的文档）是最终参考。

14.4 小结

Ptolemy II 包括一个复杂的类型系统来执行类型推断与错误检查。类型系统使用 Rehof and Mogensen (1999) 提出的一个高效算法，这个算法的复杂性对于在类型约束中符号的出现次数是线性的。因此，这个算法对大型模型也是适用的。大多数情况下，类型系统使模型的构建者不需要考虑类型的使用。这样，与标准数据库中的大多数角色一样。定义一个运用在多种类型上的角色就很容易了。

补充阅读：类型约束中的单调函数

许多复杂的类型约束可以用不等式左边的单调函数 (monotonic function) 来表示。一个单调函数 f 保持它参数的顺序；即，

$$x_1 \leq x_2 \Rightarrow f(x_1) \leq f(x_2) \quad (14-2)$$

当使用单调函数时，可以定义一个类型，它以复杂的方式依赖于其他类型。比如，RecordDisassembler 角色设定了一种类型约束，它强制输入与输出相对应的每个字段都必须是同一种类型。

单调函数可以通过子类化抽象类 MonotonicFunction 并执行 getVariables 和 getValue 方法来指定。getVariables 返回函数当作参数的变量。getValue 方法返回将函数应用于它所依赖的变量的当前值的结果。

例如，ConstructAssociativeType 类子类化 MonotonicFunction。通过 getVariables 返回的变量是保存端口类型清单的变量，与 RecordDisassembler 角色的输出端口一样。getValue 方法与这些端口名字段和端口类型相匹配的记录类型。

值得一提的是，基础类 MonotonicFunction 不能保证它的子类是具有严格的单调性。如果在类型约束中使用了一个实际上不单调的函数，那么类型解析就不能确保产生一个唯一的结果。这个结果有可能取决于约束的应用顺序。

本 体

Patricia Derler、Elizabeth A. Latronico、Edward A. Lee、Man-Kit Leung、Ben Lickly 和 Charles Shelton

在信息科学中，**本体**（ontology）是指一个明确的知识组织。本体以图的形式来组织一组概念及其之间的关系。通过构造一个凌驾于特定域的本体，用户可以通过共享的方法将这些域的信息形式化。模型可以增加一个注释来解释用户如何使用一个本体。与类型签名一样基于本体的注释是一个模型文档，该文档可以解释一个模型的用途，但是主要是解释本体的域而不是系统类型。

程序或模型的**静态分析**（static analysis）是可以在编译时运行的检查。Ptolemy II 的类型检查器（详见 14 章）就是静态分析的一个例子。它推断模型中使用的数据类型并检查其一致性。实际上，Ptolemy II 类型系统就是一个本体，它是一个模型操作的数据的知识组织。本章描述的本体检查器也执行基于注释的推断，并检查一致性，但并不要求检测数据的类型。相反，本体可以用来解释多种用户定义域中的静态分析，例如：

- **单元检查**：确定数据单元是否一致的。
- **常量分析**：确定模型中什么数据是常量，什么是变量。
- **污点分析**：确定数据流中的值是否受非信任源的影响。
- **语义检查**：确定一个部件产生数据的意思是否与另一个部件使用的意思一样。

这些分析可以揭露模型的许多错误。

例 15.1 飞行器（Moir 和 Seabridge, 2008）中多油箱系统的一部分如图 15-1 所示。这个模型有 3 个角色，端口标记了名字，要么是预期的意义要么是角色之间交换的数据单位。该模型有 3 类错误：一是单位错误（units error），一个部件传递油箱油量（用加仑表示）给另一个部件（其默认用公升表示）（后者是更好的选择，因为用加仑表示的燃油数量随着温度改变，但是用公升表示的就不会这样）；二是语义错误，一个部件将中间油箱的油量传递给一个想要看尾部油箱油量的部件；三是换位错误（transposition error），有一个油量和流量互换了。

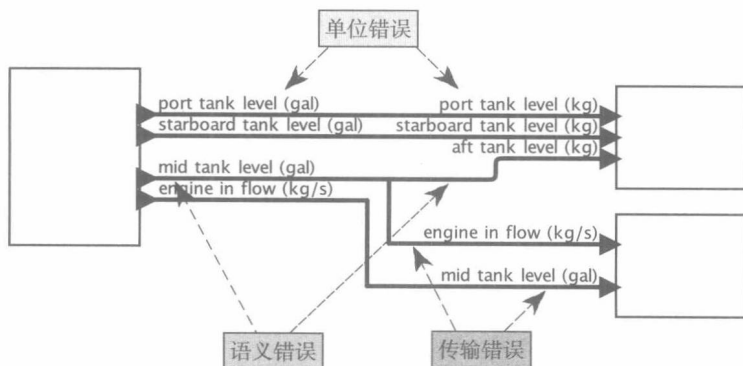


图 15-1 可以由本体系统引起的一些错误种类

这种模型错误很容易产生毁灭性的结果。本章主要对如何构造一个本体，怎样在使用它们时避免出现错误进行了概述。

15.1 创建和使用本体

本体包提供一个可以在已知模型上运行的分析系统，所以，第一步是创建一个可以运行分析系统的 Ptolemy II 模型。本节将使用一个轻量模型（rather trivial model）和一个轻量本体来说明构造本体的机制和求解器（solver）的使用方法。然后介绍一个更轻量级的本体，它对于捕获某些模型错误非常有用。

例 15.2 图 15-2 展示了拥有常量角色和非常量角色的模型。常量角色产生一系列相同值的输出。这个模型可以展示怎么产生一个用于检查模型中的信号是否是常量的简单分析。为此，首先要定义一个本体来区分“常量”和“非常量”的概念，接着定义模型使用角色的约束，最后调用求解器。

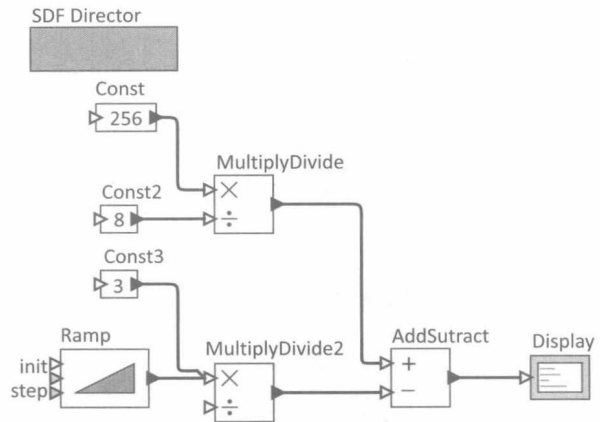


图 15-2 一个包含常量角色和非常量角色的简单 Ptolemy 模型，并包括一个用于检测信号是否恒定的本体求解器

补充阅读：本体框架背景

本章描述的本体方法是由 Leung et al. (2009) 首先提出。这种方法基于 Hindley-Milner 类型系统 (Milner, 1978) 理论、Rehof 和 Mogensen (1996) 的高效推断算法、该算法在 Ptolemy II (Xiong, 2002) 上的实现结果，以及相似数学基础概念分析的应用 (Ganter and Wille, 1998)。

值得一提的是，这个基础机制的一个扩展，是由 Feng (2009) 设计的使用本体来指导基于模型的模型转换 (model transformation)，这里启用一个 Ptolemy II 模型修改另一个 Ptolemy II 模型的结构。例如，15.1 节描述的常量分析可以指导模型最优化，使用一个 Const 角色置换所有的常量子模型。另外，Lickly et al. (2011) 提出一个无限栅格，它不仅可以用来推测一个常量信号，还能推测其值。Lickly et al. (2011) 也说明了使用无限栅格，它的其他方法，包括单位系统。他们还说明了本体怎么与结构类型（例如，一个记录）一起工作。

网络本体语言 (Web Ontology Language, OWL) 是一个广泛应用的受 W3C (the World Wide Web Consortium) 支持的语言族。OWL 本体，与本文所述本体一样，都来自于一个带有顶层和底层参数的偏序 (partial order)，不同的是，没有被约束成一个格 (lattice)。因此，Rehof 和 Mogensen 的高效推断算法不是总适用。然而，一个非常有益的机制扩展将导入和导出 OWL 本体。Eclipse 模型框架 (Eclipse Modeling Framework, EMF) 也通过类和子类概念来指定本体。许多基于 Eclipse 的工具开始支持这个框架，所以这也有利于其他工具在这方面的的发展。

15.1.1 本体创建

为了创建分析系统，第一步就是增加一个执行分析的求解器。如图 15-3 所示，拖拽一个 **LatticeOntologySolver** 角色到模型中。在这个角色中可以设置所有分析工作的细节。比如代表概念的格 (lattice)，以及角色施加在概念上的约束条件。在这个例子中，这个格将指定一个信号是否是常量，约束条件将提供一些关于哪个角色产生了常量信号或非常量信号的信息。

如图 15-3 所示，如果打开 **LatticeOntologySolver** 角色，将出现一个为创建分析所需要编辑的一个自定义程序库。分析系统至少需要一个本体，图 15-4 说明了构造它的步骤。首先拖拽一个 **Ontology** 到编辑栏，打开本体 (**Ontology**)，从本体编辑器提供的程序库中把 **Concept** 拖拽到 **Ontology** 中。

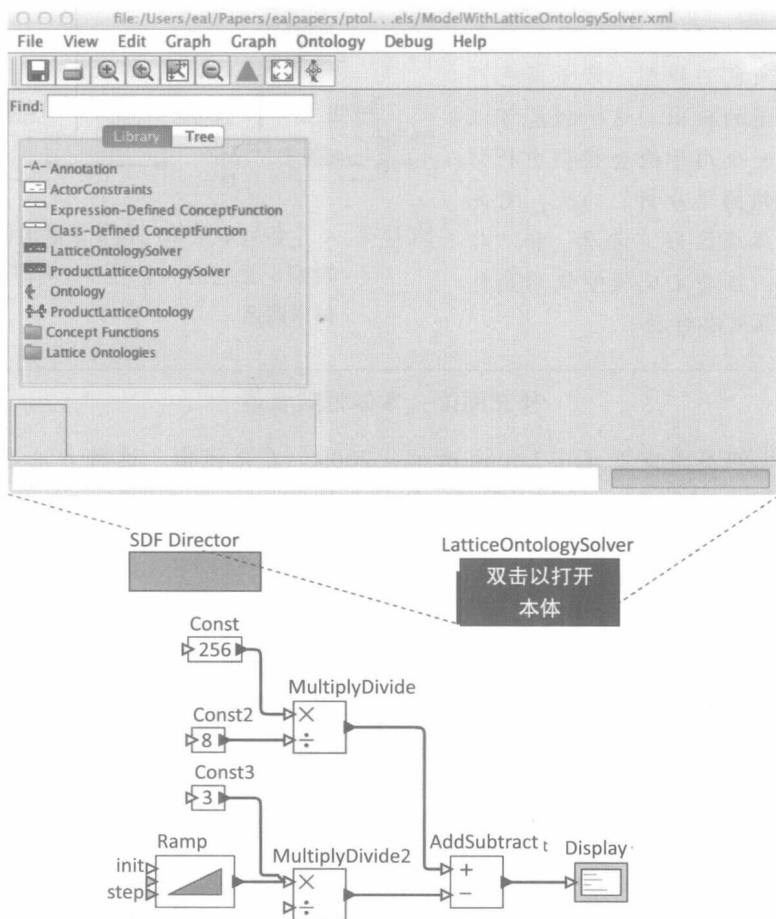


图 15-3 拥有一个空白本体求解器的模型

首先给概念 (Concept) 标注一些有意义的名字。在图 15-5 中，已经把顶部的重命名为 **NonConstant**，把底部的重命名为 **Unused**，把 **Concept** 重命名为 **Constant**。把 **NonConstant** 的参数颜色设为淡红色，检测 **isAcceptable** 参数，它移除了粗边框。这些概念将与模型中的端口相关联，当 **isAcceptable** 处于非选择状态 (概念的轮廓是粗框)，那么该模型中任何连接到这个概念的端口都是错误的。在该例中，一个端口为 **NonConstant** 不是错误的，所以将这个参数设

置为真。

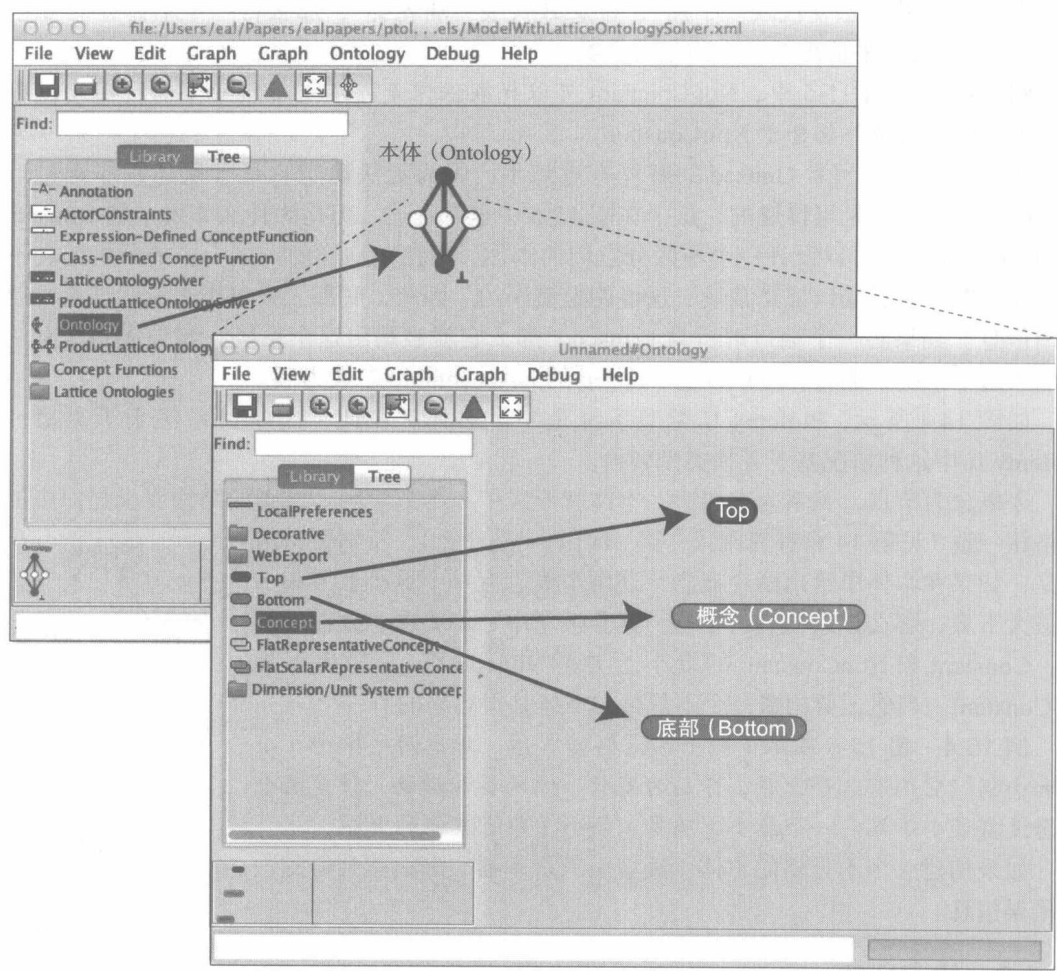


图 15-4 构造本体的步骤

此例不仅包括了 Const 和 NonConst 的概念，还明确地包括了一个 Unused 的概念。这个概念将连接到没有直接参与这个分析的端口。使用 NonConstant 来代表所有的常量信号或非常量信号，所以这个最好更名为 PotentiallyNonConst 或 NotNecessarily Const。

创建本体的最后一步是建立概念之间的关系，这可以通过使用 ctrl 键（在 Mac 中是一样的）从低 Concept 拖曳到高 Concept 来实现。在这个例子中，Constant 和 NonConstant 的关系是一个泛化关系（generalization relation）。这个关系定义了概念之间的一个次序关系（详见第 14 章补充阅读：格（Lattice）是什么，这里如果一个箭头从概念 a 指向概念 b ，那么 $a \leq b$ 。这时， $\text{Constant} \leq \text{NonConstant}$ 且 $\text{Unused} \leq \text{Constant}$ 。

这些关系的含义将随着本体的变化而变化，但是它代表一个子类、一个关系子集或“is a”关系是很普遍的。在子集解释中，一个概念代表一个集合，如果概念 A 中的所有东西都

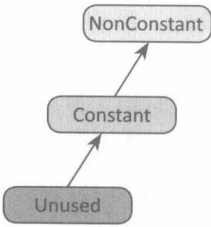


图 15-5 用于常量分析的格

在概念 B 中, 那么概念 A 小于概念 B 。“is a”关系的解释也类似, 但是不要求是集合的概念。例如, 如果概念 A 代表“小狗”, 概念 B 代表“哺乳类”, 那么将一个本体 $A \leq B$ 创建到“is a”中是合乎情理的, 小狗是哺乳类。

例 15.3 在图 15-5 中, NonConstant 可以代表任何是或不是常量的事物。因此, 有些实际上是常量的事件包含于 NonConstant。

例子中的底部元素 Unused 的解释有点棘手, 因为无法确定它是常量还是非常量。的确, 因为次序关系是可传递的, 这个在图 15-5 中也有陈述。但是为什么非要区别 Unused 和 NonConstant? 也可以设计一个不区分它们的本体, 但是在这样的本体中, 推断机制会不假思索地推断出所有端口都是常量, 这可能是错误的。因此, 底部元素将用来说明推断机制没可用信息。如果将一个端口解析为 Unused, 那么它就被踢出这个系统。如果强制所有端口都加入这个系统, 那么就应该设置 Unused 的 isAcceptable 参数为假。

如图 14.4 所示, Ptolemy II 类型系统是一个本体。在此, 次序关系代表子类型, 在 Ptolemy II 中这种情况基于无损类型转换。

令概念为节点, 关系为有向边, 本体就形成了一个数学图。该图的结构要求与对应的数学格相一致(见第 14 章补充阅读: 格(lattice)是什么)。特别要指出的是, 如果在本体中给出两个概念, 这两个概念有一个最小上界和一个最大下界, 那么该结构是一个格。这个例子中, 不要求一致性。例如, Constant 和 NonConstant 的最小上界是 Nonconstant, 最大下界是 Constant。当然, 要构造一个不是格的本体是很容易的。

例 15.4 图 15-6 展示了一个不是格的本体。考虑两个概念, 小狗和小猫。它们有 3 个上界, 分别为宠物、哺乳类和动物, 但是两个概念没有最小下界。一个最小上确界必须小于所有其他的上界。

如果构建一个不是格的本体, 接着引入求解器, 那么就会得到一个错误信息。

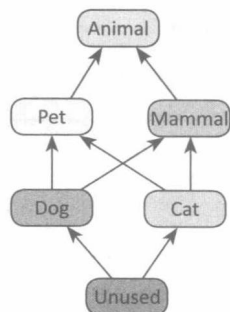


图 15-6 一个不是格的
本体

15.1.2 约束创建

既然有了一个本体, 为了能够使用它, 需要在模型中的物体与本体中的概念之间创建约束条件。最直接的方式是使用模型中的手动注释。然而, 对于庞大的模型, 这个方法不适用, 更好的方法是定义约束条件, 它广泛地应用于一个类中所有的角色, 同时在没有指定其余约束时指定默认的约束条件。首先是手动注释, 因为概念上更简单。

1. 手动注释

将模型中的物体与本体中的概念相关联的最简单方法是手动注释。手动注释使用不等式约束的形式。为了创建这样的约束条件, 在 MoreLibraries → Ontologies 库中找到约束 (Constraint) 注释, 将它拖拽至模型中。接着指定一个不等式约束条件, 格式为 `object >= concept` 或者 `concept >= object`, 这里 `object` 是模型中的一个物体(端口或参数), `concept` 是本体中的一个概念。

例 15.5 图 15-7 在图 15-3 的基础上增加了 4 个注释, 每一个注释都有如下形式

`port >= concept`

注意每一个 Const 角色的输出端口都大于或等于 Constant, 而 Ramp 角色的输出端口大于或等于 NonConstant。其实, 后面的约束强制端口为 NonConstant, 因为在本体中没有比 NonConstant 更大的。前面的约束可以通过增加额外的不等式来强制端口为 Constant, 如

Constant >= Const.output

然而, 该额外约束不是必需的。求解器将找到满足约束的最少解决方法, 因此在这个例子中, 由于 Const 角色的输出端口没有其他的约束条件, 所以无论如何它们都解析为 Constant。

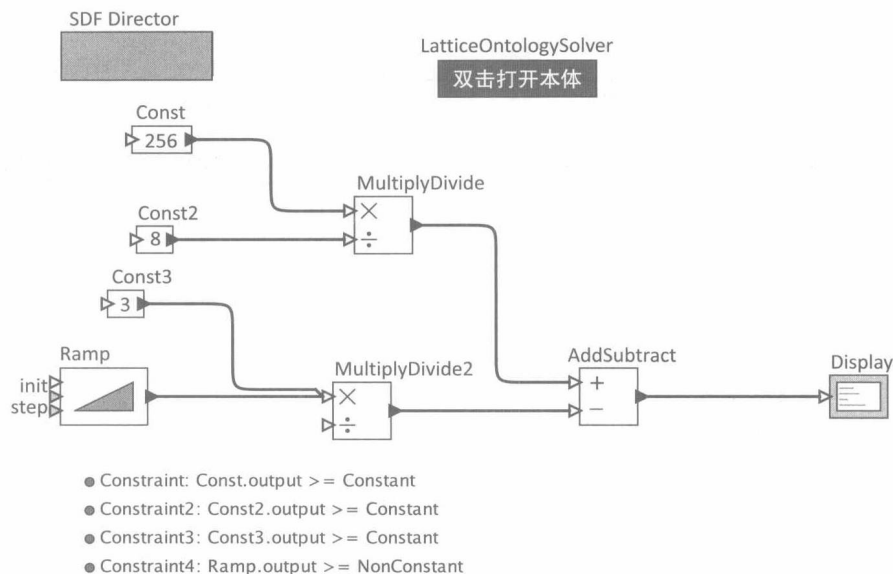


图 15-7 手动约束的模型, 使用图 15-5 中的本体

一旦创建了模型中所需的所有约束条件, 下一步就是运行这个分析系统。如图 15-8 所示, 可以通过右击 LatticeOntologySolver 并选择 Resolve Concepts 来运行。因为这是公共操作, 所以双击 LatticeOntologySolver 也能运行。

例 15.6 图 15-8 包括运行这个分析的结果。注意, 与每一个概念相连的端口都有注释。另外, 这个端口的颜色和图 15-5 的本体中的概念指定的颜色一样。正如前文所说, Const 角色的输出端口已解析为 Constant, Ramp 的输出解析为 NonConstant。更有意思的是, 下游端口也通过一个合理的方法出现转变。MultiplyDivide 的输出是 Constant (因为它的两个输入都是 Constant), MultiplyDivide2 的输出是 NonConstant (因为它的其中一个输入是 NonConstant)。这些结果是由于角色之间的默认约束, 这些约束可以在一个本体中自定义, 下面将进行解释。

2. 角色分层约束

图 15-7 中的手动注释在大的模型中变得冗长乏味。幸运的是, 有一个捷径。作为定义本体的一部分, 可以为将要关联的特定角色类的所有实例指定约束条件。图 15-9 说明了应该怎么做。在 LatticeOntologySolver 内, 为希望指定默认约束的每个角色增加一个 ActorConstraints 的实例。

设置 ActorConstraints 的 actorClassName 参数为受约束角色的完全限定类名, 使得 ActorConstraints 的名字和图标都与它约束的角色的类相匹配。

例 15.7 图 15-9 展示了将 ActorConstraints 的两个实例拖拽至 LatticeConstraintsSolver。顶部的一个使 actorClassName 参数被设置为 ptolemy.actor.lib.Ramp, 其为 Ramp 角色的类名 (为了查看一个角色的类名, 就将其拖入 Vergil)。

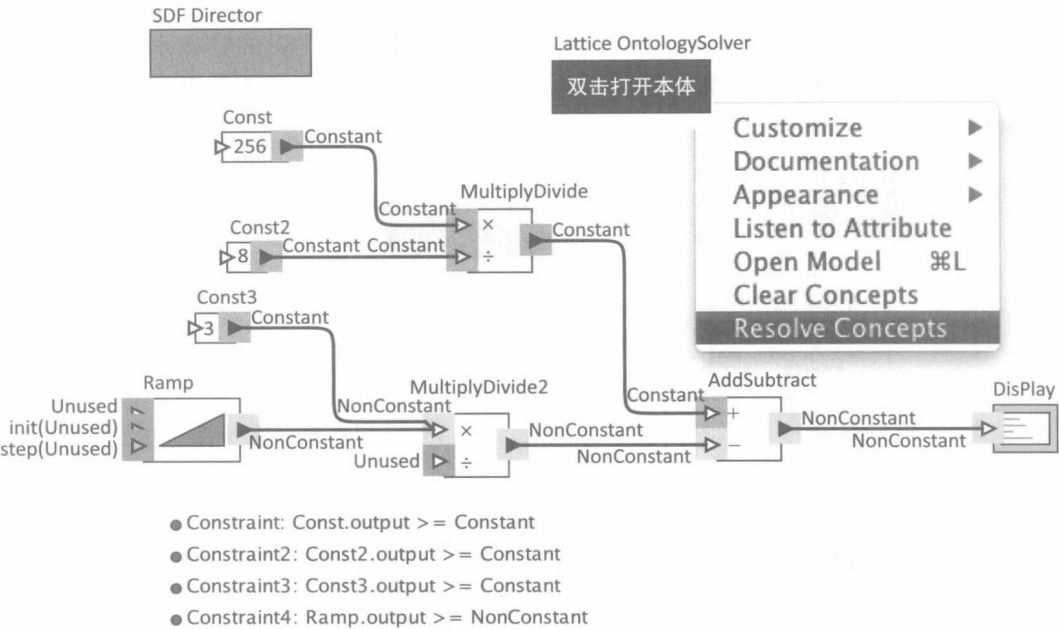


图 15-8 执行一个分析系统

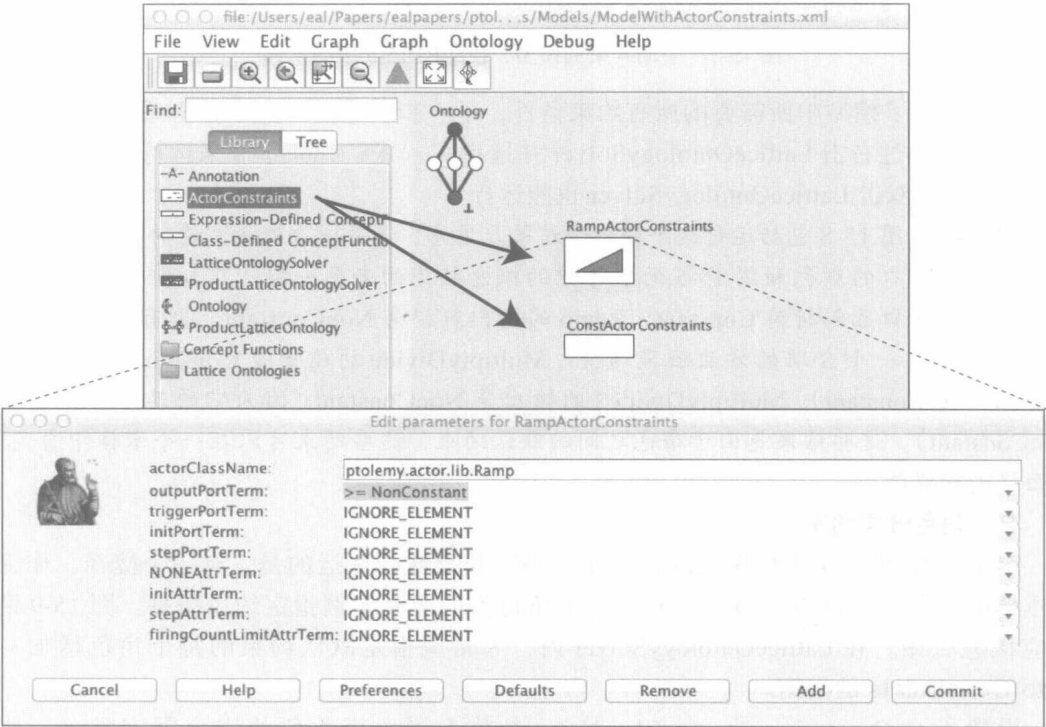


图 15-9 给本体增加角色约束

一旦设置了类名，重新打开参数对话框，一个新的参数集合就会显现出来，一个用于属于角色类实例的每一个端口，一个用于角色的每一个参数。

例 15.8 图 15-9 展示了 Ramp 角色的参数。这里，可以看到约束条件设置为 output 端

口必须大于或等于 NonConstant, 所有其他约束条件设置为 IGNORE_ELEMENT。一旦一个相似的约束增加到 ConstActorConstraints 组件中来约束 Const 实例的输出端口 \geq Constant, 那么图 15-7 的底部的 4 个约束就不再需要了。它们将被移除, 分析的结果就将与图 15-8 一样。

与端口或参数相关联的约束可以采用下面任何一种方式:

- NO_CONSTRAINTS (默认)
- IGNORE_ELEMENT
- \geq concept
- \leq concept
- $==$ concept

默认情况下, ActorConstraints 角色将给每一个端口和参数设置约束 NO_CONSTRAINTS。这就意味着, 角色的实例允许它们的端口和参数与任何概念相关联。在这个本体中, 关联总是产生 Unused, 所以让所有约束处于默认 NO_CONSTRAINTS 状态, 除了输出端口。

3. 默认约束

注意在图 15-8 中, 不只是 Const 和 Ramp 角色的输出解析为适合的概念, 下游角色也是同样, 包括 MultiplyDivide 和 AddSubtract。这些是怎么发生呢?

对于分析系统而言, MultiplyDivide 角色和 AddSubtract 角色的行为是一样的。只给出常量输入, 它们产生一个常量输出, 但给出任何非常量输入, 它们产生一个 (可能的) 非常量输出。根据本体格 (见图 15-5), 输出概念总是大于或等于输入概念。换句话说, 输出应该约束为大于或等于所有输入的最小上界。结果证明这种推断方式是很常见的。因此, 这就是所有角色的默认约束。没有显式约束的角色将继承这个默认约束。这就意味着, 可以省略指定任何 Actorlonstrains, 因为对于剩下的角色全局默认约束已经足够。

15.1.3 抽象解释

识别信号是否常量是抽象解释的一个特别简单的形式 (Cousot and Cousot, 1977)。在抽象解释中, 将变量区分类为更抽象的形式, 比如变量的值是否随时间变化, 而不是实际进行变量值的计算。可以用本体来系统地应用更复杂的抽象, 确定如变量是否总是正的、负的或 0。这个可以用来揭示设计中的错误, 同时可以通过移除不必要的计算来优化设计。

例 15.9 图 15-10 中的模型产生一个 0 的常量流。如果像以前一样在这个模型中应用 Constant-NonConstant 分析系统, 那么可以确定输出是常量。但不能确定输出是一个 0 常量流。

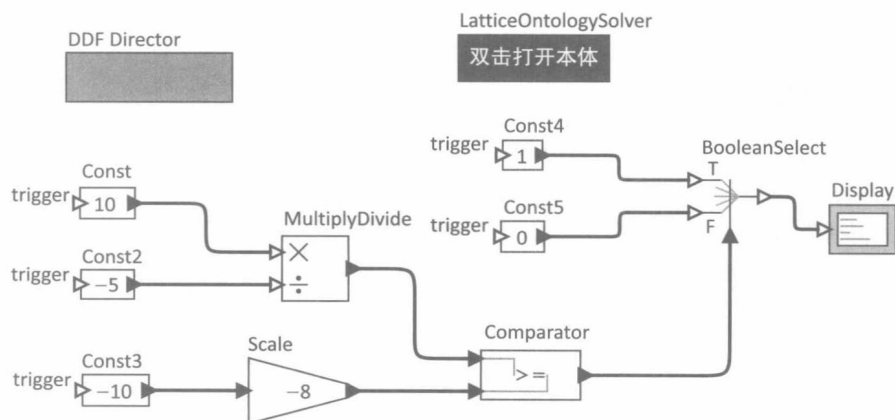


图 15-10 产生只有 0 的模型

为了解决该问题，可以使用更多如图 15-11 所示的更精心设计的本体。该本体用于将数值变量抽象为正，负或零。此外，用来区分变量是否为常量。对于布尔数值变量，如果变量是常量，那么它还需要确定常量为 true 还是 false。使用适当的角色约束，该本体可以用来产生如图 15-12 所示的结果，它确定输出是 0 常量流。

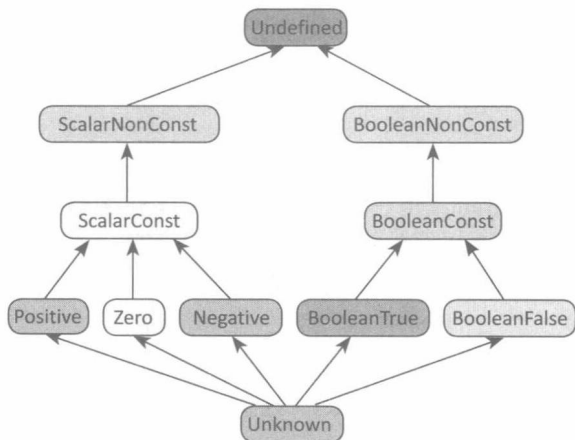


图 15-11 追踪数值变量的符号和布尔变量值的本体

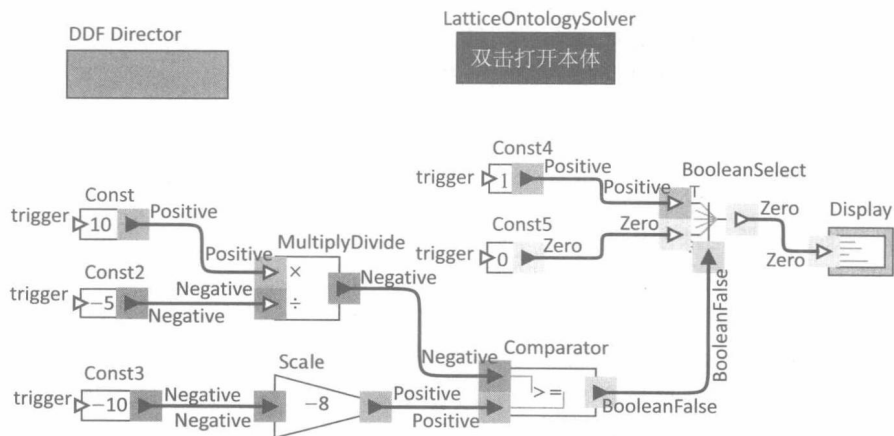


图 15-12 将基于图 15-11 模型的本体的分析系统应用于图 5-10 模型的结果

15.2 错误查找和最小化

本节将讨论怎么使用分析系统来识别错误，并讨论什么工具有助于改正错误。首先，仔细考虑图 15-13 中的本体。这个本体对物理多维建模。它区分时间、距离、速度和加速度的概念。下面将说明使用合适的角色约束，来推断这些维度的属性。

例 15.10 图 15-14 展示了维度推断模型的一小部分。这是一个有巡航控制系统的汽车模型，其中输入是理想速度，输出是加速度、速度和位置。在该模型中，速度变化量除以时间时，结果就是加速度。当加速度乘以时间时，就是速度变化量。当速度乘以时间时，就是距离。这个分析依赖于算术运算和积分器的约束。

图 15-13 中的本体明确地包括了 Unknown 和 Conflict 的概念。Unknown 与 Conflict 之间

的不同之处很微妙，值得一提。Conflict 表示分析系统发现了给定信号没有任何维度的情况。因此，分析系统发现了该模型的一个维度冲突。由于这个原因，在这个本体中，Conflict 的 isAcceptable 参数设置为 false，产生的结果在图 15-13 中用粗边框显示。另外，如果任何端口解析为 Conflict，那么运行分析系统就会报告一个错误。

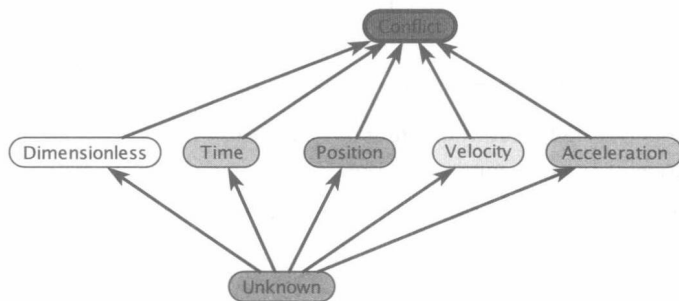


图 15-13 分析物理尺寸的本体

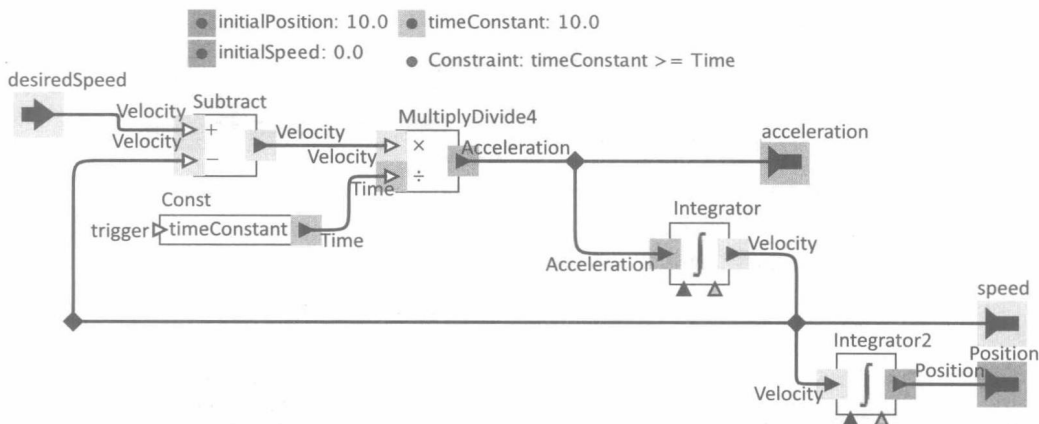


图 15-14 显示有意义尺寸推理模型的一部分

在图 15-13 所示本体中，Unknown 表示分析系统不能对给出信号任何结论的情况。这意味着，根据已知假设不能进行性能分析。这里的 Unknown 起着与图 15-5 中 Unused 类似的作用。当一个端口解析为 Unknown 时，就可以确定这个模型中没有足够的约束。这可能是一个错误，也可能不是，所以 isAcceptable 的默认值为 true。

例 15.11 图 15-15 显示了一个没有约束的模型。这里，模型用速度变量除时间来获得加速度，但是指定 Const 角色产生的时间值维度的约束被忽略了。当运行这个分析系统时，可以得到如图所示的结果。信息的缺乏贯穿整个模型。当然，可以通过增加一个手动注释来修正它，如图 15-16 所示。

通常，该约束模型将更难处理。

例 15.12 图 15-17 中显示了过约束模型的例子。这里，模型构建者错误地用时间除以距离，可能逆向操作是有意为之。结果是导致模型有冲突。

前面例子中冲突出现是因为为 MultiplyDivide 角色定义的 ActorConstrains 是本体的一部分（见图 15-9）。另外，本体给了 outputPortTerm 如下的表达式：

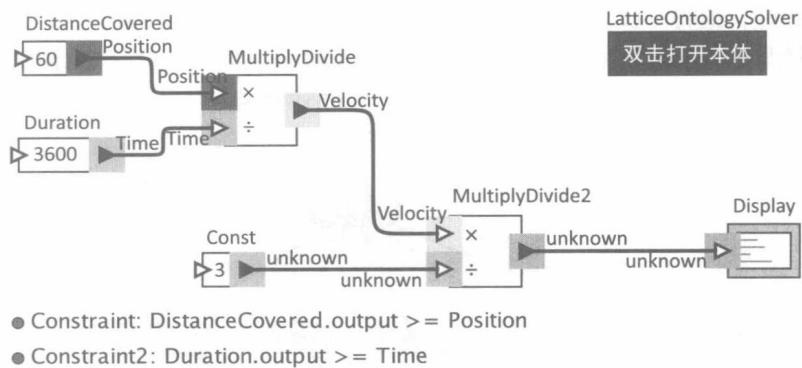


图 15-15 具有较少注释的物理维度分析。运行分析揭示这里需要额外的注释

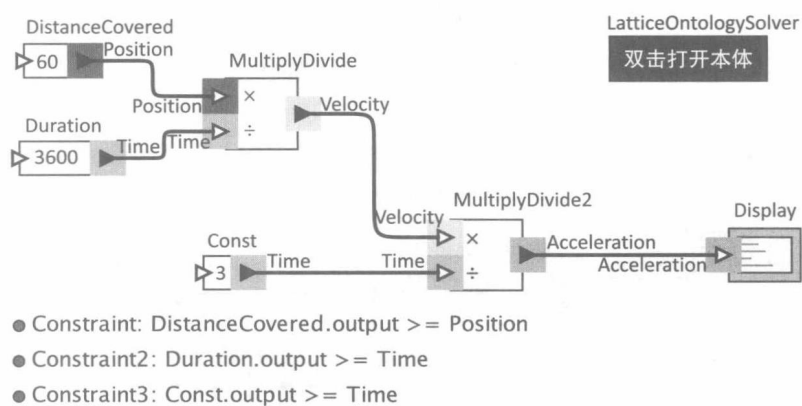


图 15-16 增加一个允许完整分析的额外约束

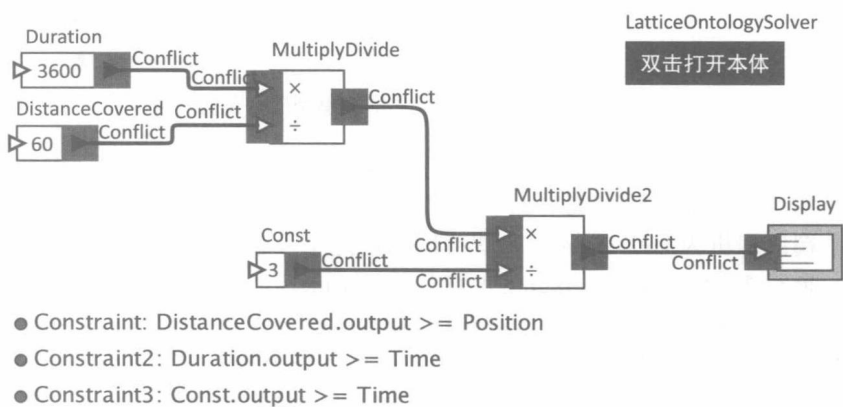


图 15-17 模型中带有尺寸冲突错误的例子。运行对该模型的分析说明整个模型处于冲突中

```
>=
(multiply == Unknown || divide == Unknown) ? Unknown :
(multiply == Position && divide == Time) ? Velocity :
(multiply == Velocity && divide == Time) ? Acceleration :
(multiply == Position && divide == Velocity) ? Time :
(multiply == Velocity && divide == Acceleration) ? Time :
(divide == Dimensionless) ? multiply :
Conflict
```

该约束是把输出概念作为输入概念的函数。例如，如果 multiply 输入是 Position，而 divide 输入是 Time，那么这个表达式计算的是速度。如果 divide 输入是 Dimensionless，那么这个表达式对任何它要乘的数进行计算。如果 multiply 是时间且 divide 是 Position，那么其结果为 Conflict。

在图 15-17 的例子中，注意冲突同时向上游和下游传播。在该例中，设置 LatticeOntologySolver 的 solverStrategy 为 bidirectional^①。这个参数默认值为 forward，这意味着如果从输出端口到输入端口之间有一个连接，那么与输入端口相关联的概念约束为大于或等于与输出端口相关联的概念。当参数值设置为 bidirectional 时，就要求两个概念是相等的。设置为 bidirectional，约束在模型中向上游传递与的向下游一样容易。所以，尽管它应用在维度分析中很合理，但是也很难区分是模型的哪个部分导致了错误。

默认情况下，求解器会找到满足所有约束的最小解^②。因此，分析系统处理冲突信息的方法就是将信号提升到冲突概念的最小上界（或者当计算最大定点时是最大下界）。

如图 15-17 所示，分析系统能够正确检测错误，但是仍存在一个问题。与相对包含错误的无约束情况不同，在这种情况下，Conflict 通过整个模型传播，这使得模型很难确定错误源在哪里。在这个简单的例子中，不难发现错误，但是随着模型的增长，确定错误源就非常难了。

为了解决该问题，提出了一个错误最小化算法。该算法在 DeltaConstraintSolver 角色中实现，它是 LatticeOntologySolver 的一个子类。可以在 MoreLibraries->Ontologies 库中找到。除了之前使用的 Resolve Concepts 外，DeltaConstraintSolver 还在上下文菜单中提供 Resolve Conflicts 项。

现在，假设有一个模型，它至少有一个信号解析为不可接受的解，由 Resolve Conflicts 实现的算法发现这些约束的一个子集，这些约束都有这样的属性：移除任何额外的约束不会产生任何错误。如果突出显示在这些约束下运行分析的结果，那么只会突出显示模型中包含错误的那些部分。实际上，如图 15-17 所示，与全局分析相比，突出显示的结果展示了这个模型的许多错误，这个改进的算法只突出显示包含错误的那些部分，如图 15-18 所示。在这个例子中，只有 Duration、DistanceCovered 和 MultiplyDivide 角色（都是 Const 的实例）被突出显示，因为它们导致了错误。没有突出显示的角色（这个例子中是 Const、MultiplyDivide2 和 Display）没有产生错误，所以模型构建者在寻找错误源时没必要考虑它们。

在这个例子中，标记为错误的端口就是错误发生的地方，但通常这没什么意义。唯一有用的就是在所有突出显示的信号中表示这里有个错误。这通常也意味着，所有突出显示的信号都需要检查以便能够找到错误源。之所以能够使用这个方法是因为几乎所有的没突出显示的角色都是可以被忽略的。

① 因为双击 LatticeOntologySolver 将运行这个分析系统，所以不能使用双击来访问该参数。相反，按住 alt 键的同时单击它来访问该参数。

② 这可以通过将 LatticeOntologySolver 的 solvingFixedPoint 参数从默认的 Least 改为 greatest 来改变。在这种情况下，求解器将在满足所有约束的格中找到最大解。

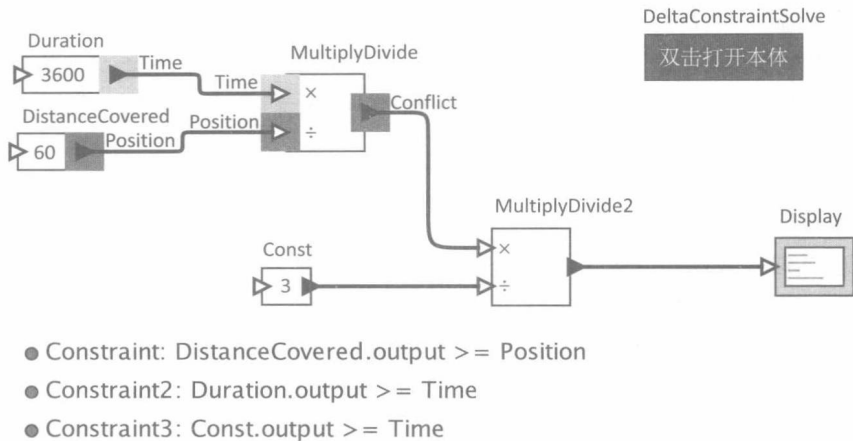


图 15-18 通过错误最小化算法可以更方便地找到错误

15.3 单位系统创建

前面章节介绍了分析系统，该分析系统使用统一的方法来检查系统的维度，这样可以避免一些错误，比如通过整合速度与预期来得到加速度的值。一个相似但是更麻烦的错误是，当两个信号有相同维度但不同单位（如英尺与米，或磅与千克）时。由于这是一个更微妙的问题（结果可能有很小比例的偏离），所以自动检测这些属性的类型可能会得到更好的结果。13.7 节描述了一个 Ptolemy II 的内置单位系统。但一个单位系统只是另一个本体，这些本体框架可以用来创建指定的单位系统。

15.3.1 什么是单位

为了讨论什么是单位，首先讨论一个单位与另一个单位有什么不同。有两种方法可以区分单位的不同。对相同的内容使用不同的测量方式，比方英尺和米，或者用完全不同的量进行测量，比如英尺和秒。第二种类型的差异其实是本体不同维度（dimension ontology）的捕获，所以我们测量单位的方法是类似的，并扩展为用单一的维度处理不同的单位。

Ptolemy II 支持创建一个与其他本体一样具有相同自由度的单位系统本体。用户可以选择适合分析当前模型的单位和维度，从而使得本体能够执行预期的分析。因为已经讨论了一个有 Position、Velocity 和 Acceleration 等概念的物理维度本体，所以将继续使用一个有相同维度的单位本体，如图 15-19 所示。然而，使用 DerivedDimension 和 Dimensionless 的实例来构造本体，而不是只使用简单的 Concept 的实例。这些都是指定的支持单位的概念，它们可以从本体 editor 提供的 Dimension/Unit System Concepts 子库中选择（见图 15-4）。

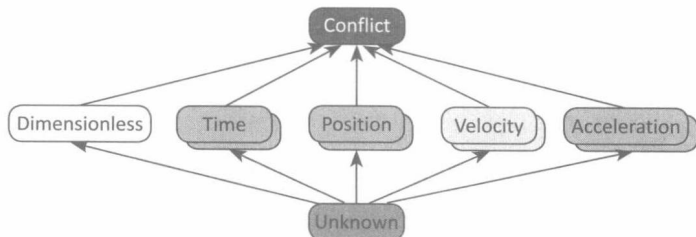


图 15-19 为物理单位创建的单位系统

如图 15-19 所示, DerivedDimension 的图标不同, 不是一个双重椭圆, 它实际上是一个代表概念 (representative concept), 代表一个概念簇族的一个单对象。DerivedDimension 有一些需要设置的参数和需要增加到单位系统里的参数。这里突显出它们的图标, 但为了从头建立一个单位系统, 还需要借助说明文档。

15.3.2 基本维度和推导维度

来自单位系统的尺寸分成两类, 基本维度 (base dimension) 和推导维度 (derived dimension)。基本维度是不能够再细分的维度, 如时间。而推导维度可以用其他的维度表示。例如, 加速度可以由 Position 和 Time 表达。哪个维度是基本维度或推导维度是没有硬性规定的。在设计构造模型时, 通常不将加速度定义为基本维度而设为推导维度, 并不是因为技术方面的原因。当定义基本维度时, 用户只需要指定该维度的单位。这可以通过给 DerivedDimension 概念增加参数来实现。

例 15.13 图 15-20 显示了增加到图 15-19 中的 Time 维度的参数。这里, 用户在一个维度内指定单位之间的比例因子, 根据需要进行命名以使得计算更清楚。

Time 维度是一个基本维度。推导维度的定义稍微有点儿复杂, 因为推导维度的定义稍微指出它从哪些维度推导而来。推导维度不仅要指出这个维度是怎么从其他维度推导而来的, 还要指出它的每一个独立单位是怎么从其他维度推导而来的。

```
secFactor: 1.0
hrFactor: 3600*secFactor
dayFactor: 24*hrFactor
sec: { Factor = secFactor }
ms: { Factor = 0.001*secFactor }
us: { Factor = 1E-06*secFactor }
ns: { Factor = 1E-09*secFactor }
minute: { Factor = 60*secFactor }
hr: { Factor = hrFactor }
day: { Factor = dayFactor }
yrCalendar: { Factor = 365.2425*dayFactor }
yrSidereal: { Factor = 31558150*secFactor }
yrTropical: { Factor = 31556930*secFactor }
```

图 15-20 指定 Time 基本维度的示例

例 15.14 图 15-21 显示了图 15-19 中的 Acceleration 维度的说明。这里, 第一行说明 Acceleration 是从 Time 和 Position 基本维度构建的, 加速度的单位为 position/time^2 。其余的说明是加速度的独立单位与距离和时间的单位相关。

```
dimensionArray: { {Dimension = "LengthConcept", Exponent = 1}, {Dimension = "TimeConcept", Exponent = -2} }
LengthConcept: Position
TimeConcept: Time
m_per_sec2: { LengthConcept = {"m"}, TimeConcept = {"sec", "sec"} }
cm_per_sec2: { LengthConcept = {"cm"}, TimeConcept = {"sec", "sec"} }
ft_per_sec2: { LengthConcept = {"ft"}, TimeConcept = {"sec", "sec"} }
kph_per_sec: { LengthConcept = {"km"}, TimeConcept = {"hr", "sec"} }
mph_per_sec: { LengthConcept = {"mi"}, TimeConcept = {"hr", "sec"} }
```

图 15-21 Acceleration 推导维度加示例

指定单位的主要好处是可以推导乘、除和积分的约束条件, 这将在许多角色中用到

15.3.3 维度之间的转换

如果单位使用不一致就会产生错误。然而, 不是所有的单位错误都是相同的。使用不同维度进行信号交换而导致的单位错误说明连接的模型必须改变, 但是使用相同维度进行信号交换而导致的单位错误可以通过转换单位来修复。虽然技术上是可以实现这种类型的自动转换, 但 Ptolemy 没有这样做。单位错误是建模中的错误, 模型构造者应该发现这些模型错误。

按照这种理论,为了能在相同维度的两个单位之间进行转变,必须给模型增加一个 UnitsConverter 角色^①。这个转换发生在分析阶段中(角色给输入输出端口的单位创建约束时),也发生在执行阶段中(角色从一个单位到另一个单位执行线性转换时)。由于本体知道组件之间的转换因子,所以模型构建者只需要指定转换的单位,而不需要指定转换逻辑来手动转换。

如图 15-22 所示,UnitsConverter 角色的参数包括 unitSystemOntologySolver,它引用单位系统分析的 LatticeOntologySolver 的名字(在此名字为 DimensionAnalysis)。由于只可能在相同维度的单位之间进行转换,所以这个维度只需要在 dimensionConcept 参数中指定一次,并在 inputUnitConcept 和 outputUnitConcept 参数中分别指定独立的输入输出单位。

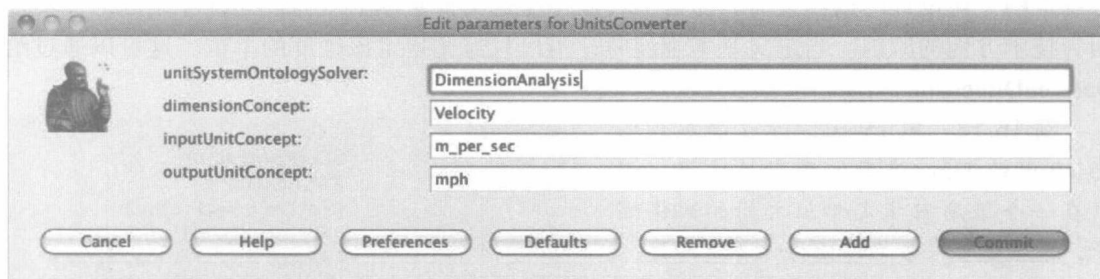


图 15-22 使用 UnitsConverter 要求设置相关求解器的名字和输入输出单位

例 15.15 假设有一个模型产生了单位错误,如图 15-23 所示,这里的速度单位为米每秒而不是英里每小时。通过增加一个如图 15-24 所示的 UnitsConverter,并设置参数使 Velocity(速度)从 _mph 转变为 m_per_sec(如图 15-22 所示),可以创建一个在运行时传递单位分析和执行转换的模型。由于本体分析不要求在运行一个模型前传递,所以图 15-23 中没有 UnitsConverter 角色的模型版本仍可以运行。但是,它会产生一个错误的输出值,因为它将 Const 中速度值的单位视为 mph,而原本是 mps。

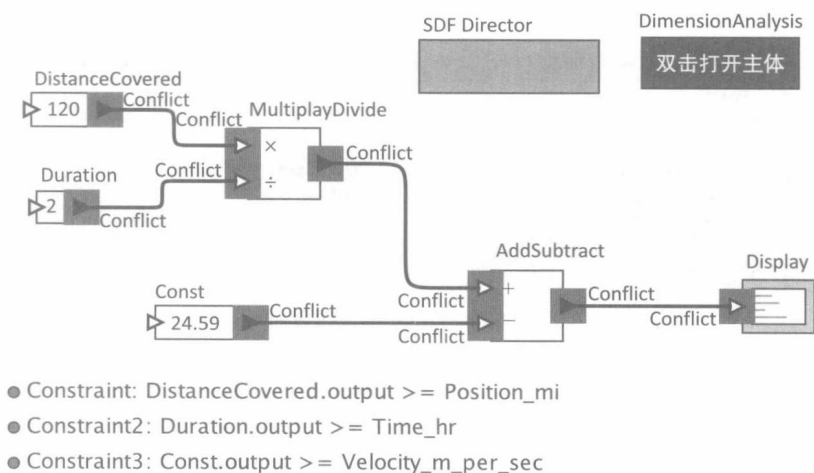


图 15-23 在没有转换的情况下,增加一个英里每小时为单位的量给一个以米每秒为单位的量,在整个模型中产生冲突

① 它可以在 MoreLibraries → Ontologies 中找到。

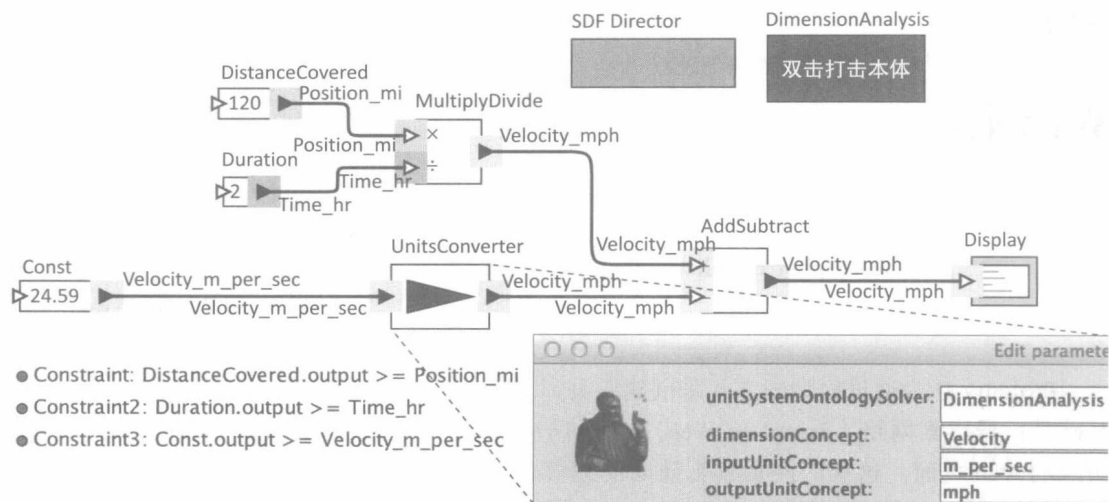


图 15-24 使用 UnitsConverter 角色将米每秒转换为英里每小时的模型

15.4 小结

构建大的、异构模型的主要挑战是确保各组件的正确性。组件通常由不同的人设计，所以它们的假设也是不同的。自定义特定域的本体提供有力的方法使这些假设更加清晰。然而，在实际应用中，这样的本体用起来很不方便，因为它们通常要求模型构建者进行广泛的注释，标记模型中每一个元素的本体信息。本章描述的基础结构使用一个非常有效的推断算法，它可以有效减少将本体应用到模型中的要求。因为大部分本体的关联都是推断的，所以它需要的注释比一般要少得多。

然而，特定域的本体和约束条件会变得十分复杂。这里假设本体库、约束条件和分析将建立并反复使用。这个对于单位系统和维度系统当然是可能的，但对于构造特定产业或特定应用分析的函数库来说它也是很有可能的。因为这些分析是简单的模型组件，所以它们很容易在企业内共享模型。

更有趣的是，简单本体可以系统地合并为更复杂的本体，包括一个被多体引用的约束条件。例如，图 15-11 中的本体可以分解为两个简单的本体：一个是数值型，一个是布尔型。一个产品本体可以由这两个简单的本体进行定义。感兴趣的读者请阅读 Lickly (2012) 的 Combining Ontologies 章节。

Web 接口

Christopher Brooks、Edward A. Lee、Elizabeth A. Latronico、Baobing Wang
和 Roxana Gheorghui

Ptolemy II 为通过模型的方式来创建网页和 Web 服务提供了一个灵活的机制。该机制的基础是：**导出到网络**（export to Web），可以简单地将一个模型转换为一个可用 Web 浏览器访问的 web 页面。这样的网页具有访问便捷模式构建和执行说明文档丰富的特点。不需要安装任何软件就可以使用它共享模型信息或模型执行信息，因为只需要一个普通的 Web 浏览器就足够了。更有趣的是，该机制具有可扩展性和个性化定制功能，它允许创建复杂的页面。你可以将超链接或 JavaScript^①中定义的行动与模型中的图标相关联。这种定制可以是模型中的个别图标也可以是模型中的图标集。

在本章中描述的更高级的机制将模型转化为 Web 服务。执行模型的机器变成了一个 Web 服务器，模型定义服务器如何响应来自因特网的 HTTP 请求。一个 Web 服务，可完成 Ptolemy II 模型所有能完成的功能。当然，需要格外谨慎的是要确保这样的 Web 服务没有给该 Web 服务器带来不可接受的安全漏洞。

16.1 导出到网络

为导出一个模型到 Web，选择 [File→Export→Export to Web]，如图 16-1 所示。这一操作将打开一个对话框，使得用户能够选择一个目录（或创建一个新目录），该目录中将生成一个名为 index.htm 的文件、一些图像文件和一些子目录。当执行导出时，一个图像文件将显示模型的所有组成。此外，还有一个用于每个打开绘图窗口的图像文件。而且，对于在导出时打开的每个复合角色，将出现一个子目录。

导出对话框的选项如下所示：

- **directoryToExportTo**：该目录用来存放 Web 文件。如果没有给出目录，则在为模型存放 MoML 文件的相同目录下创建一个新目录。该新目录将以与模型名字相同的名字命名，可以用任何特殊的字符来替换但必须保证文件名是合法的。
- **backgroundColor**：图像模型将使用的背景色。默认情况下，它是空白，这意味着图像将使用模型具有的任何背景颜色（通常为灰色）。但是，白色是一个很好选择，如图 16-1 所示。
- **openCompositesBeforeExport**：若该选项为 true，则模型中的复合角色在导出前被打开。每个复合角色还将输出到各自的网页，并在最上层图像中创建超链接来允许导航到浏览器中的网页。如果只想在导出中包括一些复合角色，你可以手动打开。只

① 默认情况下，导出到网络设备使用 JavaScript 来显示角色的参数。JavaScript 可能在用户浏览器中不能使用。为了在浏览器中使用 JavaScript，详见 <http://support.microsoft.com/gp/howtoscript>。

有打开的窗口才能包含在导出中。

- **runBeforeExport**：若该选项为 **true**，则该模型在导出前运行。这将对打开图形窗口产生副作用，然而它包括在导出中。若你只想在导出中包括一些图形窗口，那么运行模型，关闭那些不需要的模型。在导出中只包括打开的图形窗口。
- **showInBrowser**：若该选项为 **true**，则一旦导出完成，在默认浏览器中将显示生成的页面。
- **copyJavaScriptFiles**：若该选项为 **true**，则导出页面将包含一些附加的文件，使得页面不依赖于任何因特网文件。文件包括 JavaScript 代码以及影响网页交互性和外观的图像文件。默认情况下，没有包括这些文件，这些文件可以通过网站以下获得：<http://ptolemy.org>。

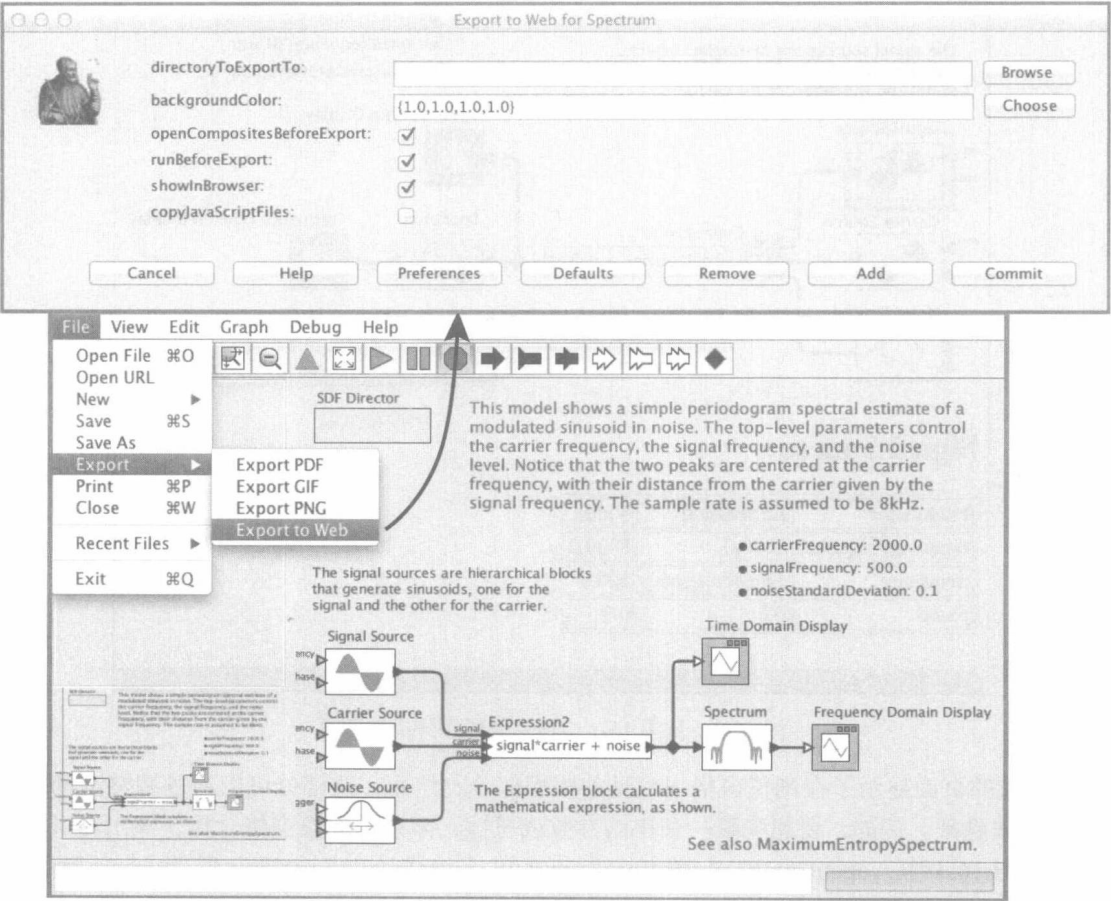


图 16-1 导出模型到网络的菜单命令

对于图 16-1 中的例子，由 Safari Web 浏览器显示的页面如图 16-2 所示。这些页面显示了一些导出到网络的默认行为。网页的标题显示在页面的顶部，默认情况下是模型的名字。此外，图 16-2 中的图像，当鼠标停在 Signal Source 角色上时，显示其轮廓。当鼠标停在一个角色上时，默认情况下，在页面的底部显示一个带有角色参数值的表格，如图 16-2 所示。

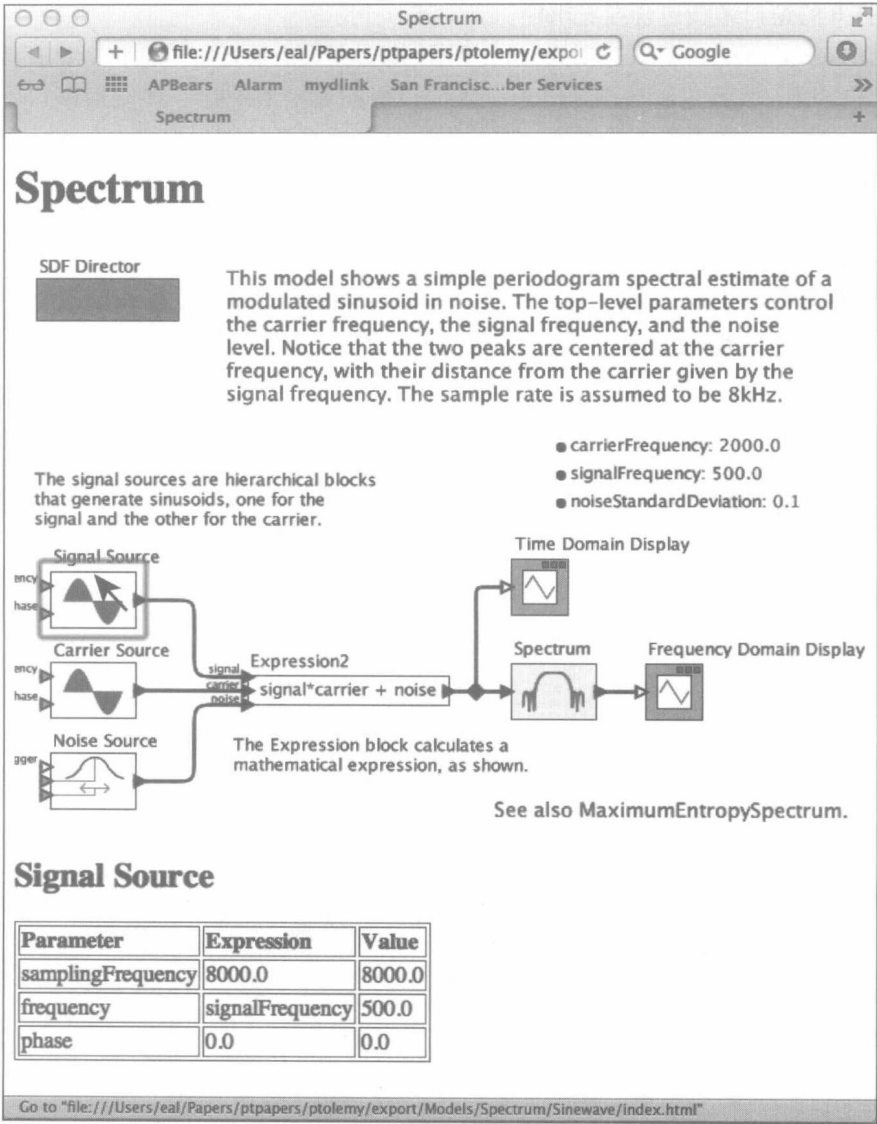


图 16-2 从图 16-1 中显示的模型导出的网页

生成的页面显示了在浏览窗格中可见的模型部分。然而，这部分可以通过调整浏览窗格 (Pane) 来隐藏。例如，要想隐藏一长串的参数或属性，简单地调整窗格，执行导出即可。

在图 16-1 中，openCompositesBeforeExport 和 runBeforeExport 都设置为 true (默认是 false)。因此，在导出前执行模型，打开图像窗口。创建到图像窗口的超链接，在 Web 页图形上点击图形角色就会显示图像，如图 16-3 所示。此外，模型中的复合角色，如 Signal Source、Carrier Source 和 Spectrum，都有到显示复合内部结构页面的超链接。

以上所有功能都可以定制，将在下面讨论。

16.1.1 导出定制

如图 16-4 所示，Utilities → WebExport 库提供属性，当它被拖入模型中时，这些属性能定义导出的网页。本节对库中的每一项进行解释，如图 16-4 左边所示。用户可以右击

(或在 Mac 上单击)，选择 Get Documentation 来查看属性文档。与另一个在 UML 类图有关的属性如图 16-5 所示。

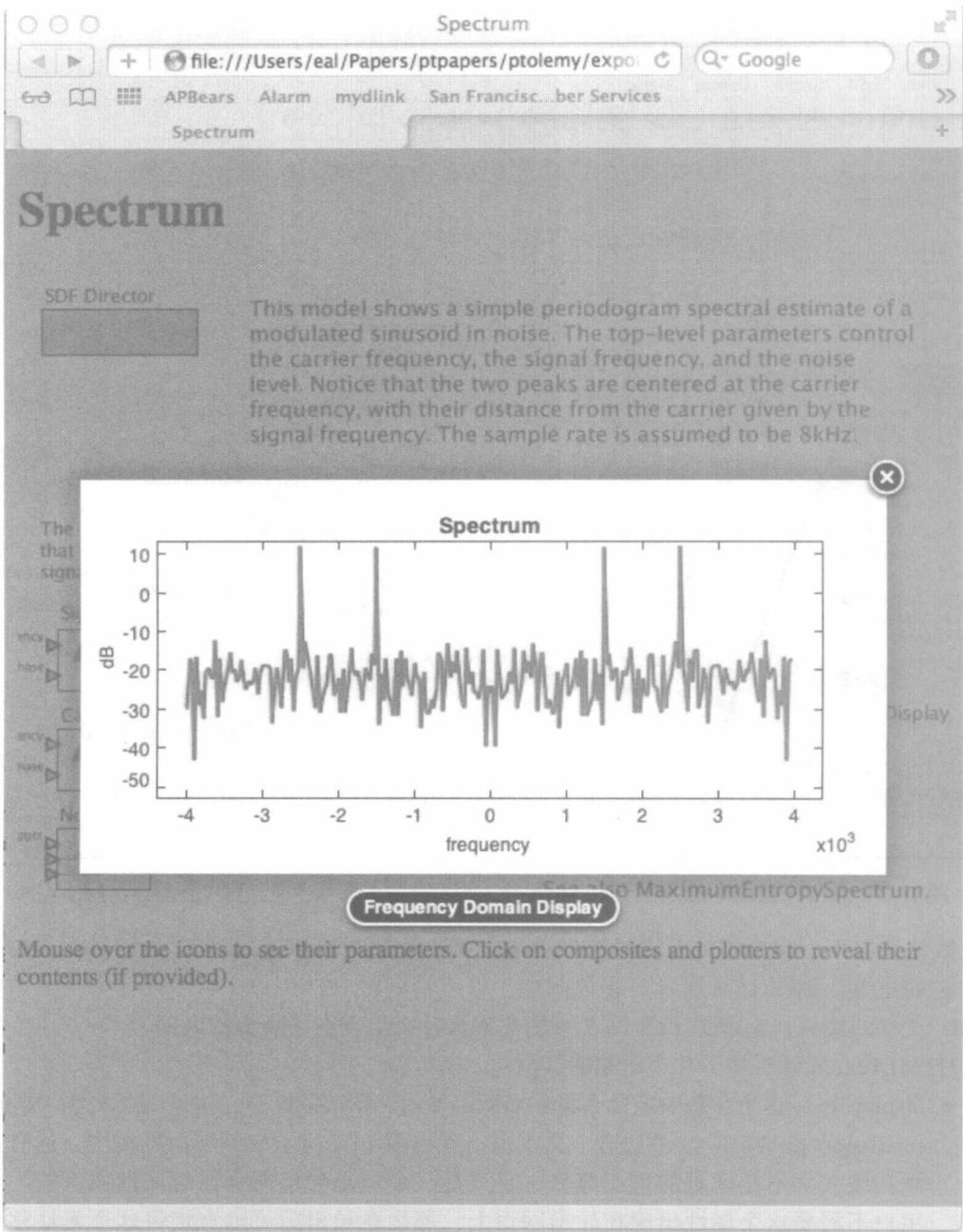


图 16-3 单击图 16-2 中 Frequency Domain Display 角色，显示运行该模型产生的图形

1. HTMLText：添加文本到页面

HTMLText 属性将 HTML 文本插入到由 Export to Web 导出页面上。将属性拖到一个模型的背景上，如图 16-4 所示，双击它的图标，指定要导出的 HTML 文本。为了在 HTML 页面上指定要包含的文本，双击 HTMLText 属性的图标（默认情况下它是一个文本图标阅

读“Content for Export to Web”), 如图 16-6 所示。用户可以输入要导出的任意文本, 包括超链接和内容格式化的指令等任何 HTML 内容。图 16-7 显示的网页包括了图 16-6 中指定的文本。

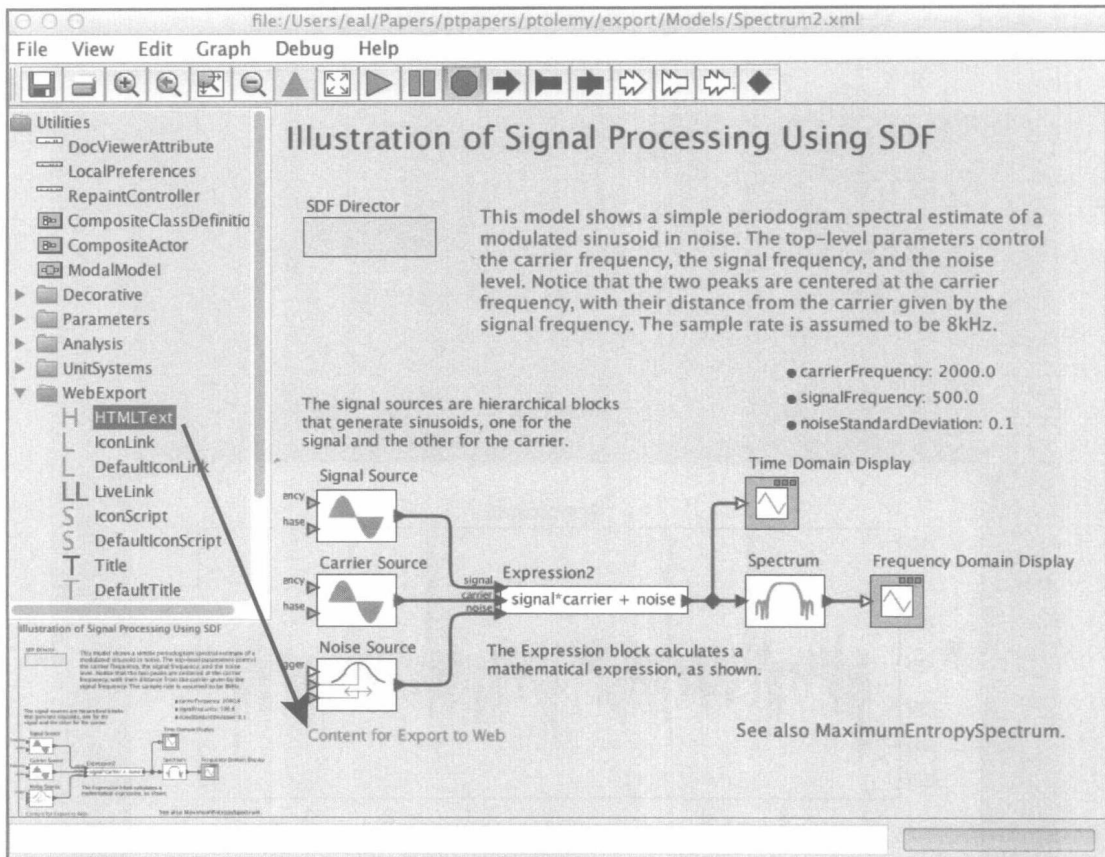


图 16-4 Utilities/WebExport 库提供属性, 当它被拖入模型中时, 可以定制导出网络页面

默认情况下, 文本将放置在模型图像的前面, 但用户可以通过设置 `textPosition` 参数来改变它的位置, 如图 16-6 所示。在该图中, 可以看到将 `HTMLText` 属性设置为将文本放在 HTML 文件的最后, 它解释了图 16-7 中的文本为什么出现在了页面的底部。

`HTMLText` 属性有如下几个定制选项:

- `displayText`: 这个参数决定什么将在模型上显示。默认情况下, 这是一个文本“Content for Export to WEB”。注意这个文本也出现在图 16-7 所示中的导出网页中, 这有点奇怪。该文本不是模型的重点部分, 它只是一个定制导出网页面的属性占位符。如果用户不需要这个属性出现在导出网页上, 那么在导出前可以直接将这个属性移出页面。或者将 `displayText` 设置为一个空字符串, 但这种方法存在缺点, 它会增加用户寻找编辑或定制导出文本属性的难度。在图 16-8 中, `displayText` 已设置为空字符串。`HTMLText` 参数仍然存在并可以选择 (图中左下方的深灰色小框就是 `HTMLText` 参数)。编辑 HTML 参数更简单的方法是右击模型的背景, 如图 16-8 所示。与其他在模型中定义参数一样, `HTMLText` 也出现在模型的参数栏里。

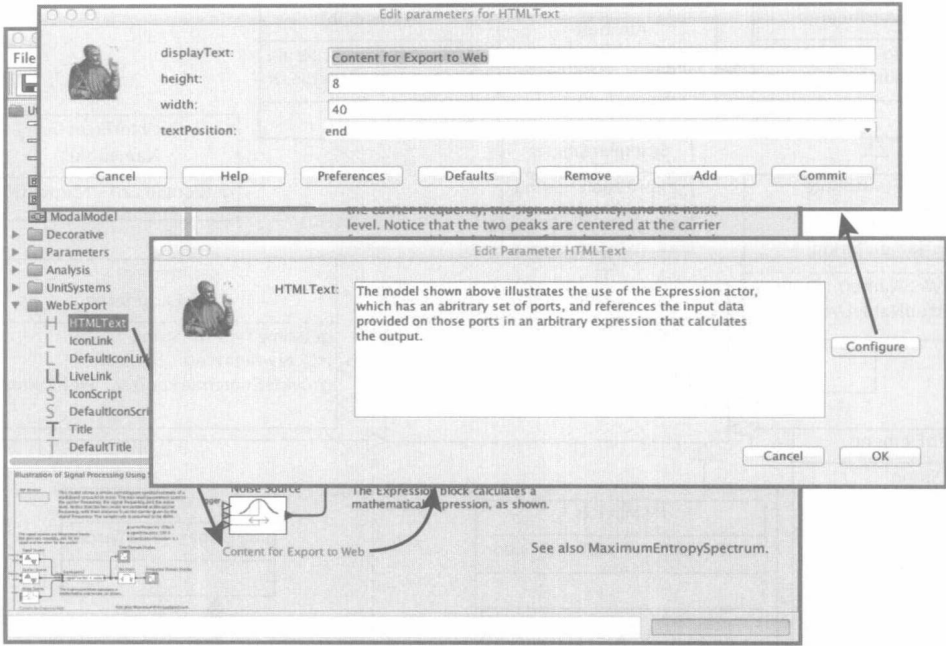


图 16-6 包含在导出网页的定制 HTML 文本的对话框

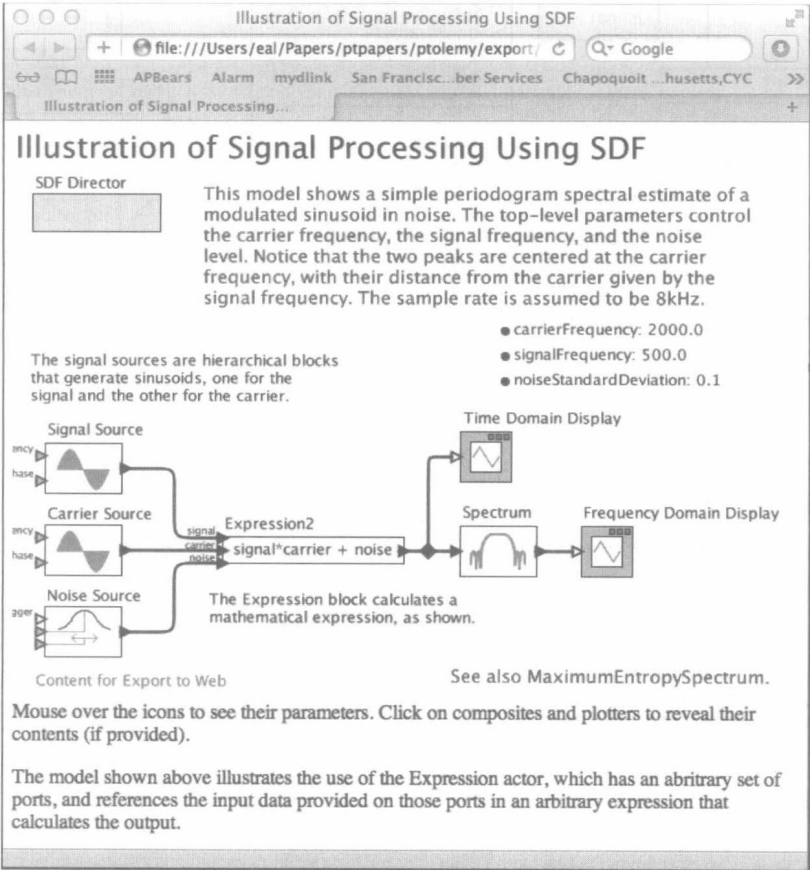


图 16-7 在图 16-4 的示例中插入 HTMLText 属性的页面，参数设置如图 16-6 所示

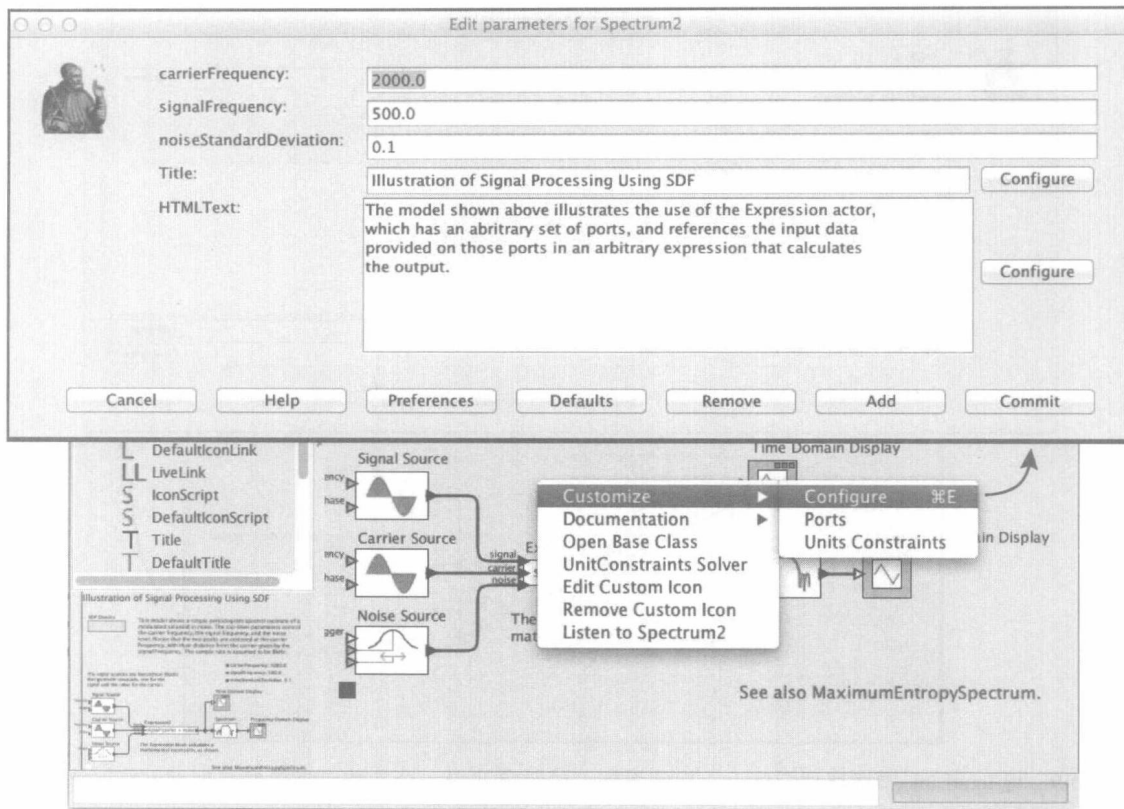


图 16-8 HTMLText 属性可以通过设置 displayText 参数为空字符串来隐藏。也可以通过右击模型背景来编辑它。注意 HTMLText 出现在了模型参数列表中

- height：指定导出文本编辑框的高度。如果你改变了这个值，关闭并重新打开对话框使改变生效。
- width：指定导出文本编辑框的宽度。如果你改变了这个值，关闭并重新打开对话框使改变生效。
- textPosition：如上所述，该参数决定导出文本的位置。内置的选项是 end（最后）、start（开始）和 head（头部）。选择“end”将文本放在导出模型图像的后面；选择“start”将文本放在导出模型图像前面；选择“head”将文本放在 HTML 页的头部。如果为 textPosition 指定任何其他值，那么系统将这个值作为文件名，并在同一个目录下创建具有该文件名的文件作为导出文件。然后指定的文本将导出到这个文件中。

2. IconLink：为图标指定超链接

在 Utilities → WebExport 库中显示的 IconLink 参数可为模型中的图标创建超链接。为使用它，直接将它从库中拖到想要链接的图标上。在图 16-9 的例子中，已经把它放置在右下角的文本注释上，上面显示“See also Maximum Entropy Spectrum”。双击该文本注释显示 IconLink 参数，这个参数可以设置为一个 URL。导出的网页面将包含一个从文本注释到指定页面的超链接。

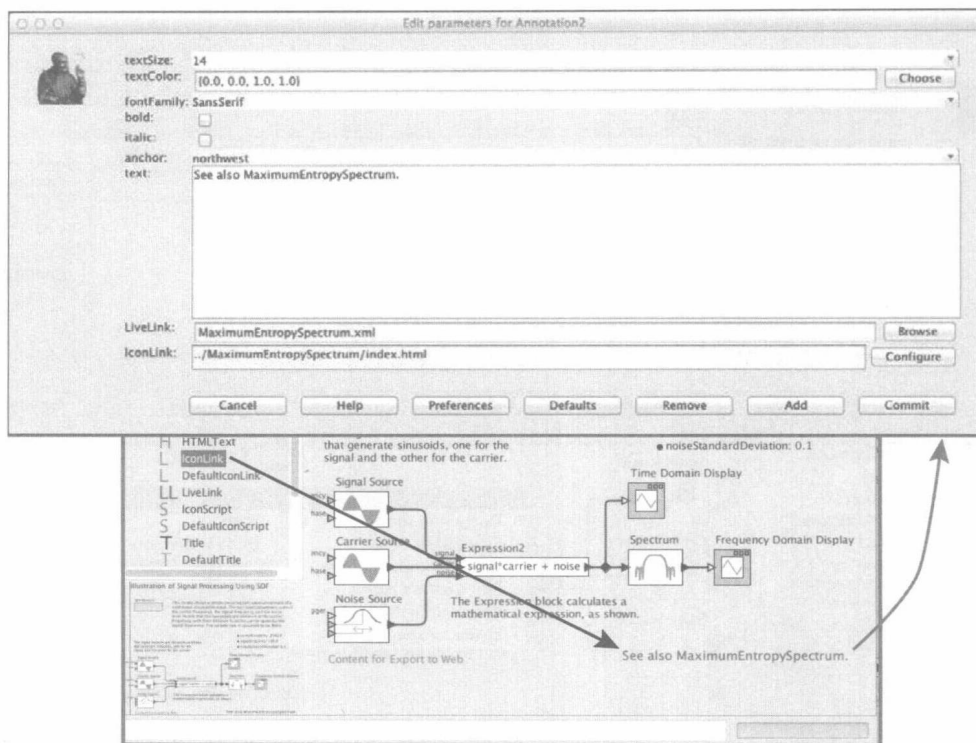


图 16-9 将 IconLink 属性拖到图标上。拖到图标上的对象获得一个参数，当模型导出到网络时该参数能够指定从图标到网页的链接。这里，导出的网页包括图标链接为 “../MaximumEntropySpectrum/index.html”

可以定制 IconLink 参数（单击图 16-9 中对话框右下角的 Configure 按钮）。参数 displayText、width 和 height 与上文中的 HTMLText 一样。一个新的参数是 linkTarget，它有 4 个可以设定的值：

- `_lightbox`：在弹出灯箱（Lightbox）中显示链接。
- `_blank`（默认）：在浏览器的新空白窗口中显示链接。
- `_self`：在同一个窗口中显示链接，替换当前页面或框架。
- `_top`：在同一个窗口中显示链接，替换当前页面。

图 16-3 是灯箱显示的例子。

此外，如果给出 linkTarget 参数任何其他值，那么这个值就假设为网页中框架的名称，且该框架变成当前目标。

3. DefaultIconLink：图标的默认超链接

在图 16-4 中，左边 Utilities → WebExport 库显示的 DefaultIconLink 参数可用于为不包含 IconLink 的模型中的任何图标指定一个默认超链接。除了 IconLink 参数外，DefaultIconLink 还有两个参数：

- `include`：这个参数可以用于限制默认应用的图标。具体来说，默认情况下可以为图标指定属性、实体或两者。
- `instancesOf`：如果非空，该属性指定一个类名。只有实现指定类的实体或属性（取决于 include 参数）才能分配默认链接。

4. LiveLink: Vergil 中的超链接

虽然不是直接与网页导出相关，但是库中还是包括 LiveLink 参数，因为它与 Iconlink 匹配得很好。如果将一个 LiveLink 实例拖到一个图标上，那么当用户双击 Vergil 中的图标时（在浏览器中单击显示导出网页的图标），可以指定打开一个文件或 URL。这不会自动在导出网页中产生一个超链接，因为通常模型需要指定不同的在 Vergil（而不是浏览器）中打开的文件或 URL。例如，Vergil 可以打开并显示 MoML 文件，而浏览器只简单地显示 XML 内容。

例 16.1 注意，在图 16-9 中，注释“See also MaximumEntropySpectrum”既包括 IconLink 实例也包括 LiveLink 实例。LiveLink 引用了一个 MoML 文件，MaximumEntropySpectrum.xml，与 Spectrum 模型存储在相同的目录中。然而，IconLink 引用了一个 HTML 文件，前提是 Spectrum 和 MaximumEntropySpectrum 都已导出网页，且这些网页在服务器上的相关位置是指定的路径，这些路径为 MaximumEntropySpectrum 提供一个到 HTML 文件的链接。

假设所有文件都适当地安排在文件系统中，那么 Vergil 超链接和网页超链接的功能是一样的。它们将打开引用的模型 MaximumEntropySpectrum。但是 Vergil 在 Vergil 中打开它，而浏览器在浏览器中打开它导出的网页。

5. IconScript: 图标的脚本操作

IconScript 参数用来与提供模型中图标相关的脚本操作。具体地说，一个操作可以与鼠标移动、鼠标点击或键盘操作相关联，该操作可以指定为 JavaScript 脚本。

例 16.2 图 16-10 和图 16-11 显示了使用 IconScript 的例子。在这个例中，IconScript 参数的两个实例被拖到 Ramp 角色的图标上。当鼠标放到导出网页上的图标上时，这些参数定制显示“I am a Ramp actor!”；鼠标离开图标时，清除显示，如图 16-11 所示。

第一个 IconScript 参数的值是 JavaScript 代码：

```
writeMyText('I am a Ramp actor!')
```

它调用了 JavaScript 程序 writeMyText，它在 IconScript 参数的 script 参数中定义为：

```
function writeMyText(text) {
    document.getElementById("below").innerHTML = text;
};
```

这个程序使用一个参数 text，将其值写进 ID 为 below 这个元素的 innerHTML 文件中。这个元素定义在 IconScript 参数的 endText 参数中，如下所示：

```
<p id="below"></p>
```

这是一个具有 ID below 的 HTML 段。该段将插入模型图像下的导出网页面中。最后，IconScript 的 eventType 参数设置为 onmouseover，它是调用脚本的结果，当鼠标移入显示 Ramp 图标的网页的区域时，如图 16-11 所示。

IconScript 的第二个实例为 IconScript2，它指定以下脚本：

```
writeMyText('')
```

当鼠标移出 Ramp 图标时，这个脚本通过相同的 JavaScript 程序清除显示。第二个 IconScript 的 eventType 参数设置为 onmouseout。

如果 IconScript 的多个实例有完全相同的 script 参数，那么这些参数的值将只在导出的 HTML 页面的头部中包含一次。因此，script 参数的值需要 JavaScript 定义。如果在模型中至少需要一次网页导出器，那么它们可以只包含这些定义一次。

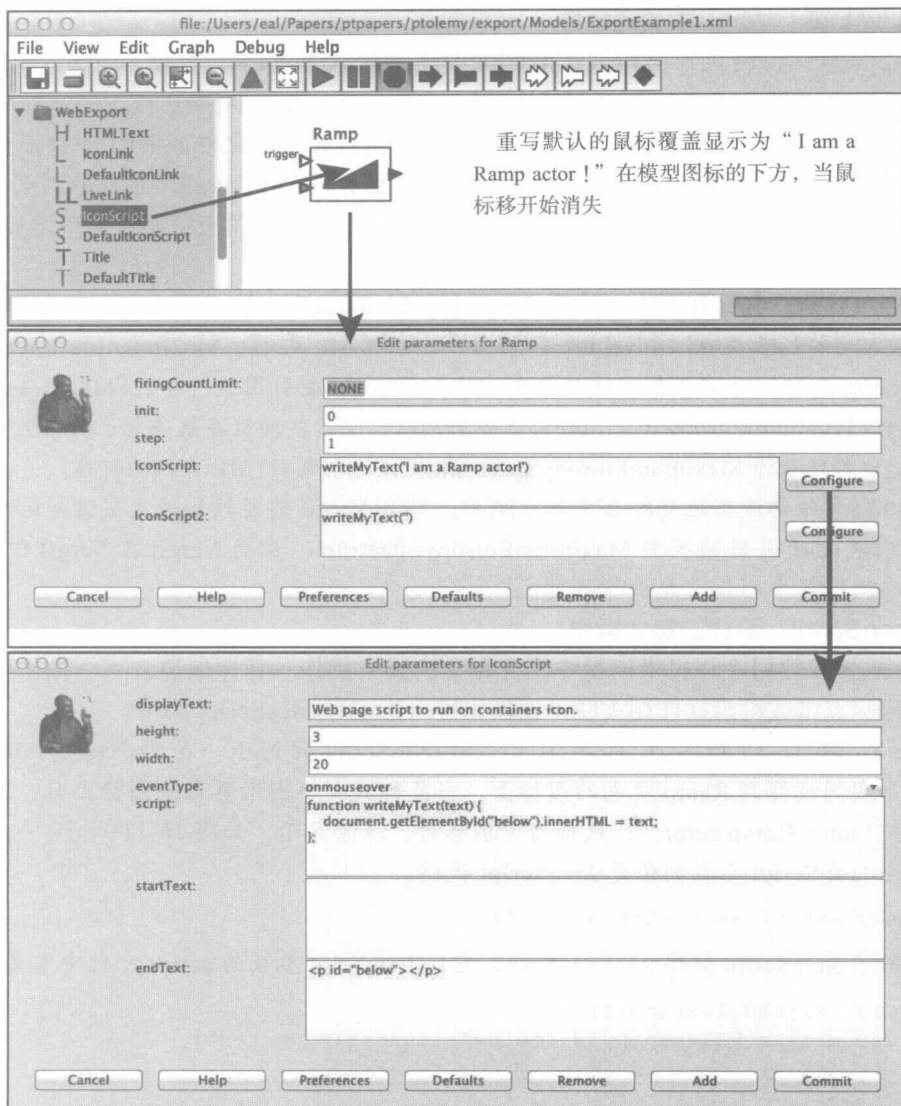


图 16-10 IconScript 参数的两个实例已经拖到 Ramp 角色的图标上。当鼠标放在导出网页的图标上时，这些参数已经定制显示“ I am a Rampactor! ”，鼠标离开图标时清除显示，如图 16-11 所示

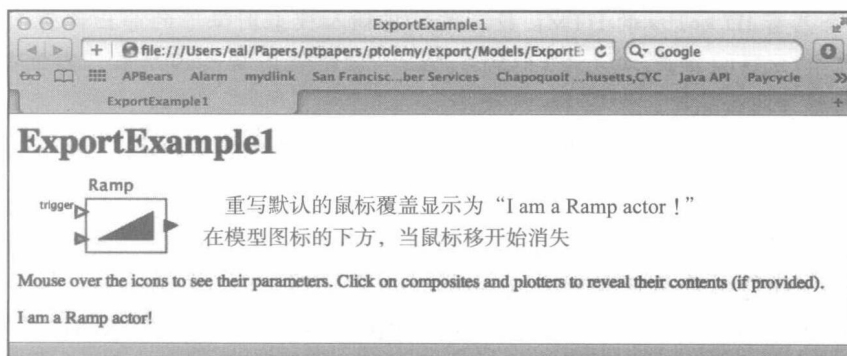


图 16-11 图 16-10 中模型导出的网页，鼠标移动到 Ramp 图标时显示的结果

6. DefaultIconScript: 图标的默认脚本操作

除了它是被拖到模型的背景上而不是图标上以外, DefaultIconScript 与 IconScript 类似。并且它能为多个图标指定操作。DefaultIconScript 的参数与 IconScript 的一样, 但与上面描述的 DefaultIconLink 一样, 它还有 include 和 instancesOf 参数, 这些与 16.1.1 节中描述的一样。

例如, 可以使用 DefaultIconScript 覆盖鼠标移开时显示的参数的默认操作, 如图 16-2 所示。

7. Title: 图标的标题

Title 参数用于定制网页上显示的标题。这个参数也作为一个标题出现在 Vergil 窗口中。图 16-7 中的标题实际上就是一个插入模型中的 Title 的实例, 默认标题改为 “Illustration of Signal Processing Using SDF”。它代替网页导出的默认标题, 它是模型的名称。这个标题也成为导出 HTML 文件的头部中定义的标题。

Title 参数的默认值是表达式

```
$(this.getName())
```

在 Ptolemy II 表达式语言中这是一个字符串参数的表达式 (参见第 13 章)。这个表达式调用容器对象中的 getName 方法, 所以显示的默认标题为模型的名称。

8. DefaultTitle: 图标的默认标题

DefaultTitle 参数用于定制模型中与每个图标相关联的标题。当鼠标在图标上来回移动时, 标题作为提示信息 (tooltip) 显示在导出网页上。与 DefaultIconLink 一样, 它包括 include 和 instancesOf 参数, 它们与 16.1.1 节描述的意思一样。这些可以用来为标题的子集指定默认的标题。

16.2 Web 服务

Ptolemy 允许模型作为 Web 服务运行。Web 服务运行在服务器上, 在因特网上通过统一资源定位符 (URL) 可以访问它。通常, Web 服务对请求做出响应, 通过网页 (一般以 HTML 格式, 超文本标记语言) 或通过以一些其他的标准因特网格式如 XML (可扩展标记语言) 或 JSON (JavaScript 对象表示法) 来提供数据。标准 Ptolemy II 库包含一个属性, 该属性可使模型转换为一个 Web 服务、一个响应 HTTP 请求的角色、一些方便构建 HTML 响应的角色, 以及访问和使用 Web 服务的模型的角色。

16.2.1 Web 服务器的架构

图 16-12 显示了一个 Web 服务器的操作。访问 Web 服务器的 URL 包括协议、主机名和端口号 (默认端口号为 80)。例如, URL `http://localhost:8078/` 表示向运行在端口 8078 的本地机器上的 Web 服务器发送一个 HTTP 请求。

补充阅读: 命令行导出

给出一个模型的 MoML 文件, 就能通过命令程序 ptweb 生成一个网页。命令的格式如下:

```
ptweb [options] model [targetDirectory]
```

“model”参数必须是一个 MoML 文件。如果没有指定目标目录，那么模型的名字将变成目标目录的名字（其中不能包含特殊字符，如果有，需要替换掉）。选项包括：

- -help: 打印帮助信息。
- -run: 在网页导出前运行模型，这样图形窗口就包括在导出中。

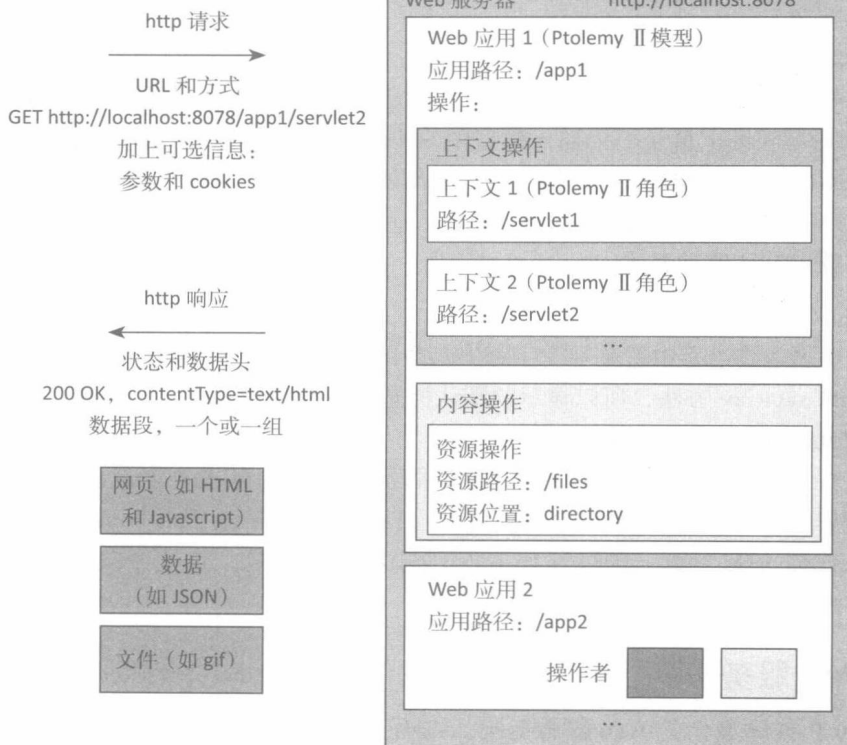


图 16-12 一个 Web 服务器拥有一个或多个 Web 应用程序。每个应用程序包含一个或多个请求处理程序。

Web 服务器接收 `HttpRequest`，并根据请求的 URL，将请求交给合适的请求处理程序。处理程序返回 `HttpResponse`

Web 服务器拥有一个或多个 Web 应用程序（或 Web 服务）。在本文的例子中，每个应用程序将由一个 Ptolemy II 模型实现，这个模型包含 Web 服务器属性的实例（见第 16 章补充阅读：Web 服务和网页的相关组件）。每个应用程序都在服务器上注册一个**应用程序路径**。应用程序将处理对 URL 的 HTTP 请求，URL 包括在主机名和端口号之后的应用程序路径。例如，如果应用程序 2 注册应用程序路径 `/app2`，那么 URL `http://localhost8078/app2` 将由应用程序 2 处理。应用程序路径可以是空字符串，这种情况下对于这个端口上该主机的所有 HTTP 请求都交给该应用程序处理。当在同一台服务器上运行多个应用程序时，每个应用程序应该有一个独一无二的**应用程序路径前缀**，这样服务器可以确定将请求提交给哪里。

每个应用程序包含一个或多个请求处理程序。在本例子中，这些处理程序是 `HttpActor` 角色的实例。每个处理程序在 Web 应用程序上注册一个**路径前缀**（这个路径前缀可以是空字符串）。（见第 16 章补充阅读：Web 服务和网页的相关组件）对 `HttpActor`，路径前缀由 `path` 参数给出。当多个处理程序在同一个应用程序中运行时，每个处理程序都应该有一个单

独的路径前缀，这样应用程序才能确定将请求交给那里。例如，在图 16-12 中，URL `http://localhost:8078/app1/servlet2` 将由应用程序 1 来处理，并将它交给已经注册前缀 `servlet2` 的 Ptolemy II 角色。如果有多个前缀匹配，那么服务器将选择最具体的前缀来处理。例如，如果一个处理程序有两个前缀，一个是空白前缀，另一个是 `/foo`，那么所有“`http://hostname applicationPath/foo /...`”形式的请求都提交给第二个处理程序，而其他格式的请求提交给第一个处理程序。

第二种类型的处理程序为资源处理程序，它也是 Web 服务提供的用来处理对静态资源的请求，如文件（Jetty 类 `ResourceHandler`）。而且，它必须有一个出现在 URL 中的前缀。例如，在图 16-12 中，URL `http://localhost:8078/foo/app1/files/foo.png` 引用了一个名为 `foo.png` 的文件，该文件存储在一个服务器的目录中，该目录用来存储 Web 服务的资源位置属性。

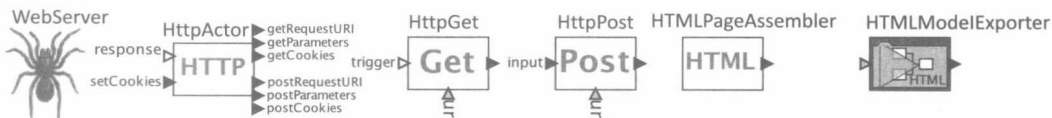
处理程序产生的响应包含一个状态码、一个头部和一个响应体。响应体是用户的内容，可以是一个网页、一个文件或者 JSON 格式的数据。状态码指示操作是否成功，如果没有成功，说明失败原因。对于 HTTP 请求，有一组标准的响应代码^①。头部包含内容格式（MIME 类型^②、内容长度、其他有用信息）。

16.2.2 构建 Web 服务

WebServer 和 HttpActor 的使用方法如下例所示。

补充阅读：Web 服务和网页相关的组件

一些对构建 Web 服务、访问网页和建立网页有特殊用途的组件如下所示。



- **WebServer**：当包含它的模型执行时，它启动 Jetty Web 服务（参见 <http://www.eclipse.org/jetty/>）。该属性传入的 HTTP 请求路由到执行 `HttpService` 界面的模型中的对象中，如 `HttpActor`。这个属性带有指定接收 HTTP 请求端口的参数，应用程序路径包括在 URL 中，该 URL 用来访问服务器、寻找请求资源的目录，以及存储临时文件的目录。
- **HttpActor**：处理与它的路径匹配的 HTTP GET 和 HTTP POST 请求的角色。这个角色和 DE 指示器一起工作。输出包含从服务器模型开始执行经过时间（以秒为单位）的请求时间戳的详细情况。角色希望它产生的每个输出，它驻留的模型都将提供一个响应 HTTP 请求的输入。
- **HttpGet**：给指定的 URL 发出 HTTP GET 请求的角色。该角色与 `FileReader` 类似，但它只处理 URL，而不处理文件。
- **HttpPost**：给指定的 URL 发出 HTTP POST 请求的角色。放置的内容由输入记录指定。

① <http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>.

② <http://www.iana.org/assignments/media-types/index.html>.

- **HTMLPageAssembler**：该角色通过在模板文件的适当位置插入输入文本来组装 HTML 页面。
- **HTMLModelExporter**：VisualModelReference 角色的扩展，它不仅显示和执行一个参考模型，而且还使用 16.1 节讨论的技术将这个模型导出到一个网页。

例 16.3 图 16-13 中的模型是一个 Web 服务，它要求用户输入一些文本，然后返回“Ptoleminized”文本，所有首位的“p”(但不包括“th”的实例)替换为“pt”。例如，“text”变成“ptext”，如图 16-14 所示。文本处理由 PythonScript 角色完成，它执行图 16-15 中的 Python 代码。

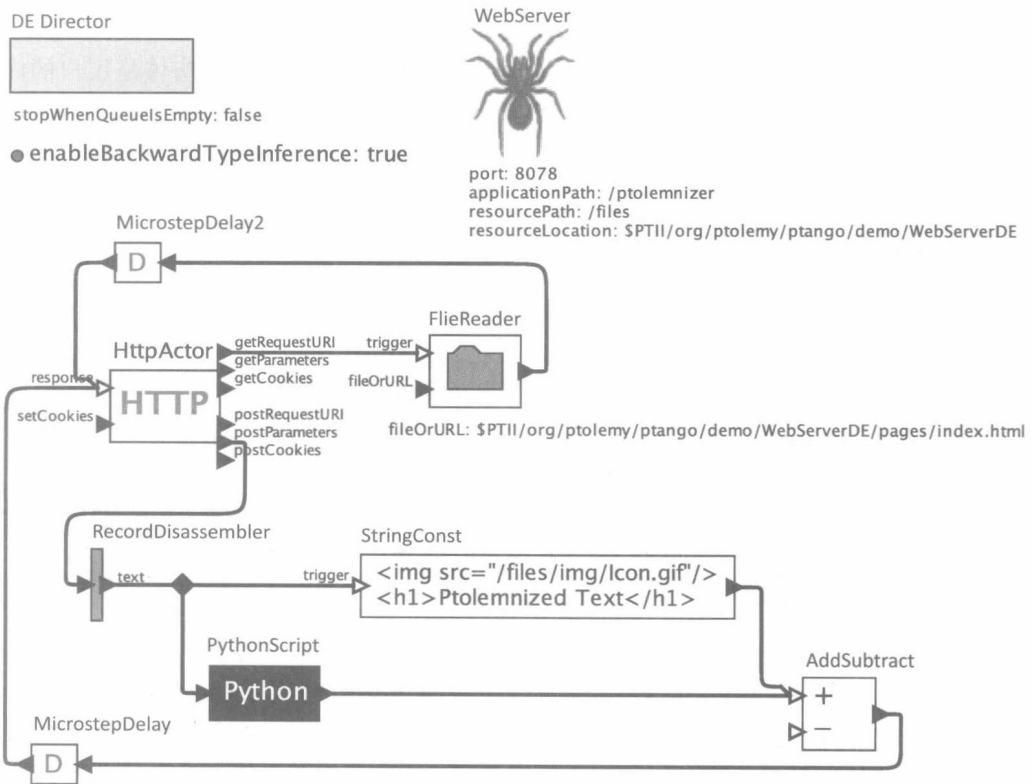


图 16-13 一个在 Ptolemy II 中执行的简单 Web 服务

首先，注意 DE 监视器的 stopWhenQueueIsEmpty 参数设置为 false。如果不是，当运行时模型就会立即停止因为没有要处理的挂起事件。其次，注意模型的 enableBackwardTypeInference 参数设置为 true。这是反向类型推理，这种情况下，使 HttpActor 的 postParameters 输出有一个称为类型字符串的“test”单个区域的记录类型。PythonScript 角色在它的输入端口指定类型字符串，因为 Python 代码需要一个字符串。

当模型执行时，Web 服务器在本地机器的端口 8078 上通过应用程序路径 /ptolemnizer 启动一个 Web 服务。因此这个服务在 <http://localhost:8078/ptolemnizer> 上。在 Web 浏览器上访问 URL 产生图 16-14 的顶部网页。这是如何实现的呢？

当 Web 服务用匹配的应用程序路径接收到一个 HTTP GET 请求时，它将请求交给 HttpActor。

角色请求指示器点火，以及指示器点火该角色时，它在 HttpActor 的 3 个输出端口的顶部产生关于 GET 请求的信息。模型使用 GET 请求的 URL 触发 FileReader 角色，它只读取本地文件系统上的文件，文件内容如图 16-16 所示。该文件的内容被发送回 HttpActor 的响应输入，然后再次点火。在第二次点火时，它与 WebServer 合作为图 16-14 顶部的响应提供服务。注意，在反馈回路中需要 MicrostepDelay 角色，通常当作 DE 模型（见 7.3.2 节）。

如图 16-14 和图 16-16 所示，服务的网页有固定的格式，按下“Ptolemy”按钮产生这种格式内容的 HTTP POST。该 POST 出现时，WebServer 再次提交给 HttpActor，在其 3 个输出端口的下面输出 POST 的细节。postParameters 端口将产生一个叫作“text”的单独字段的记录令牌。RecordDisassembler 提取该字段的值，这是由用户输入表单的文本。然后，StringConst、PythonScript 和 AddSubtract 角色构建一个 HTML 响应，它被发送回 HttpActor。这个响应产生图 16-14 底部的页面。

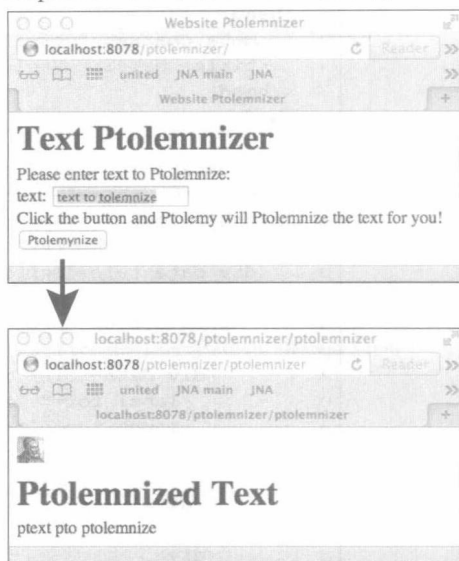


图 16-14 由图 16-13 中模型返回的网页响应一个 HTTP GET，返回的页面响应一个 POST（由按钮点火）

```

1 from ptolemy.data import StringToken
2 class Main :
3     "ptolemyizer"
4     def fire(self) :
5         # read input, compute, send output
6         t = self.in.get(0)
7         s = t.stringValue()
8         s = self.ptolemyize(s)
9         t = StringToken(s)
10        self.out.broadcast(t)
11        return
12
13    def ptolemyize(self, s) :
14        l = list(s)
15        length = len(l)
16        if length == 0 :
17            return ''
18        if length == 1 :
19            if l[0] == 't' :
20                return 'pt'
21            else :
22                return l[0]
23        if l[0] == 't' and l[1] != 'h' :
24            l[0] = 'pt'
25            i = 1
26        while i < length - 1 :
27            if l[i-1] == ' ' and l[i] == 't' and l[i+1] != 'h' :
28                l[i] = 'pt'
29                i = i + 1
30        if l[-2] == ' ' and l[-1] == 't' :
31            l[-1] = 'pt'
32        return reduce(lambda x,y: x+y, l, '')

```

图 16-15 图 16-13 中 PythonScript 角色的 Python 代码

```

1 <!DOCTYPE html>
2 <head>
3   <meta charset="utf-8">
4   <title> Website Ptolemnizer </title>
5 </head>
6 <body>
7
8 <div data-role="page" data-theme="c">
9   <div data-role="header">
10    <h1> Text Ptolemnizer </h1>
11  </div>
12  <div data-role="content">
13    Please enter text to Ptolemnize:
14    <form action="ptolemnizer" method="post" >
15
16      <div data-role="fieldcontain" class="ui-hide-label">
17        <label for="text">text:</label>
18        <input type="text" name="text" id="text" value=""
19              width="80" placeholder="text to tolemnize"/>
20      </div>
21    </div>
22
23    <div>
24      Click the button and Ptolemy will
25      Ptolemnize the text for you!
26      <br/>
27      <button type="submit" id="ptolemnize">
28        Ptolemy nize
29      </button>
30    </div>
31  </form>
32 </div>
33 </div>
34 </body>
35 </html>

```

图 16-16 图 16-13 中 FileReader 角色所读取的 HTML 代码

对 POST 的响应包括一个“img”元素（参见图 16-13 中的 StringConst 角色）。当浏览器解析该响应时，这个 img 元素将触发另一个 HTTP GET。Web 服务器将 resourcePath 参数设置为 /files，所以 img 源 URL /files/img/Icon.gif 由资源处理程序处理，而不是交给 HttpActor 处理（见图 16-12）。这个资源处理程序将搜索由 resourceLocation 参数给定的目录中名字为 img/Icon.gif 的文件。图 16-14 底部页面上的小 Ptolemy 图标就是搜索结果。

这个例子通过组合多项功能来构建 Web 服务。它使用 HTML 构建了一个交互式网页，用 Python 去处理用户提交的数据。实际上，对于一系列不同软件组件，Ptolemy 模型是作为一个协调器来服务的。

16.2.3 使用 cookie 在客户端存储数据

cookie 是一个小数据块（包含生存周期和可见性的信息），它由 Web 浏览器存储于客户端，并将随后的 HTTP 请求返回给 Web 服务器。Web 服务可以在客户端使用 cookie 存储状态。例如，Web 服务可以使用 cookie 来记住已经登录的用户。永久性 cookie 存储指定时间段（包括无限期的）的内容，而会话 cookie 存储的内容会随着浏览器窗口的关闭而丢失。

HttpActor 为来自客户端浏览器的会话 cookie 提供基本的获取和放置支持。具体来说，

HttpActor 有一个 `requestedCookies` 参数，该参数的值是一个字符串数组。它指定由 Web 服务设置或得到的 cookie 的名称。它还有一个输入端口 `setCookies`，这个端口接受一个记录，该记录将值赋给每个已命名的 cookie。最后，通过输出端口 `getCookies` 和 `postCookies` 为每次 HTTP 的 GET 和 Post 请求提供一段记录。

例 16.4 图 16-17 中的模型使用了 cookie。该模型中的 Web 服务使用 cookies 记住通过一系列 HTTP 访问的客户端的身份。图 16-18 中的页面说明该服务如何响应初始的 HTTP GET、作为 cookie 存储客户端“Claudius Ptolemaeus”身份的 HTTPPOST，随后的 HTTP GE 和删除 cookie 的 HTTP POST。

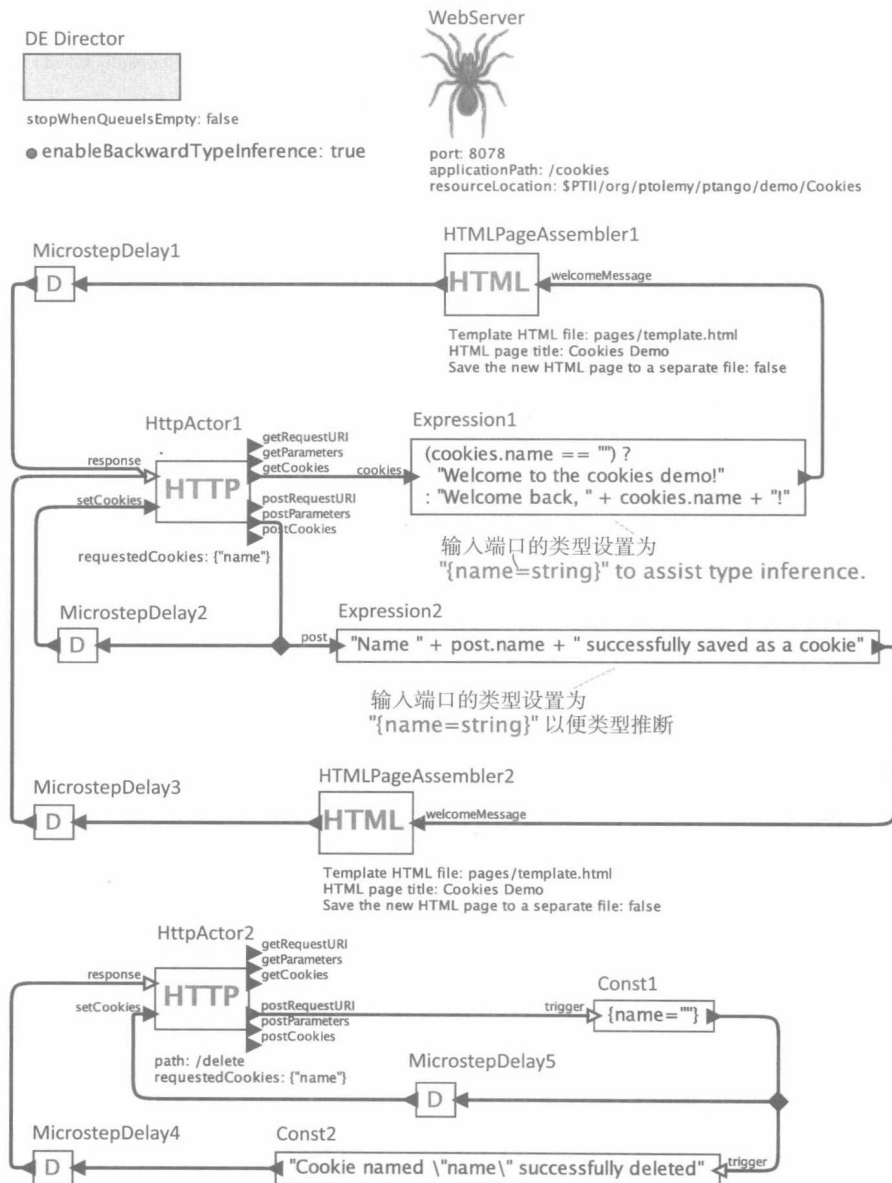


图 16-17 一个用来获得、设置和删除客户端 cookie 的模型

模型有两个 HttpActor 的实例。第一个，标记为 HttpActor1，有默认的 path（路径）参

数，它匹配所有的请求。第二个，标记为 `HttpActor2`，有路径设置 `/delete`，所以它将用 `http://localhost:8078/cookies/delete` 形式的 URL 处理请求。

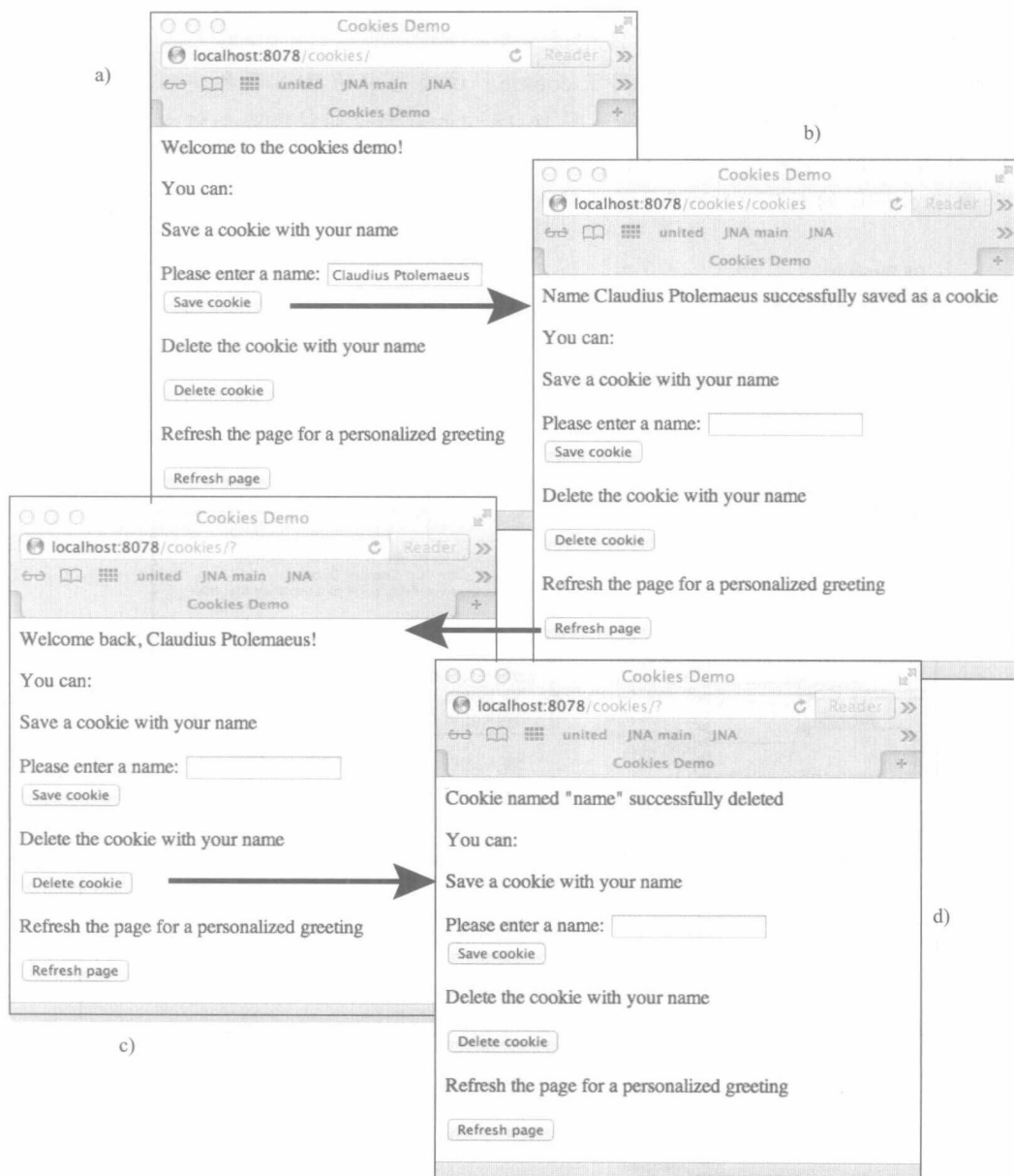


图 16-18 由图 16-17 中的模型创建的一系列网页

在 `HttpActor` 的两个实例中，参数 `requestedCookies` 设置为 `{ "name" }` 和有一个字符串的数组。它通知 `HttpActor` 检查标记为 `name` 的 cookie 的传入 HTTP 请求。`HttpActor` 在它的 `getCookies` 或 `postCookies` 输出端口上用 cookie 提供的标签 `name` 和值产生一个记录。如果没有找到 cookie，值是一个空字符串。

注意，`HttpActor` 角色总是用 `requestedCookies` 中指定的字段产生一个记录，所以下游模型总是可以假设记录具有指定的字段。例如，在图 16-17 中名为 `Expression1` 的 `Expression`

角色使用语法 `cookies.name` 提取该记录的 `name` 字段。如果字段值是一个空字符串，那么模型生成一个如图 16-18a 所示的通用欢迎消息。否则，生成定制的面，如图 16-18c 所示。

在图 16-18a 中，从最初的页面开始，用户可以指定一个名称并用这个名称来保存一个 cookie，它产生如图 16-18b 所示的响应。这些是使用一个带有参数 `name` 的 HTTP POST 完成的。注意在图 16-17 中，`postParameters` 输出端口反馈到 `setCookies` 输入端口，所以对个 HTTP POST 的响应将在浏览器中用 POST 提供的任何值来设置一个 cookie。

单击“Refresh page”按钮产生另一个 HTTP GET，它现在产生定制的面，如图 16-18c 所示。

单击“Delete cookie”按钮，给 `http://localhost:8078/cookies/delete` 发送一个 POST 请求。该请求映射到 `HttpActor2`。响应分为两部分。首先，`Const1` 发送一个带有标签 `name` 的记录和一个空字符串值给 `HttpActor2` 上的 `setCookies` 端口。`HttpActor2` 将它解释为删除 cookie 的请求。注意，由于这个执行，`HttpActor` 角色将任何带有空字符串的 `RecordToken` 标签解释为删除带有标签 cookie 的请求。因此，丢失的 cookie 相当于一个具有空值的 cookie。此外，该模型将生成一个确认 cookie 删除的响应，如图 16-18d 所示。

组装网页

例 16.4 和图 16-17 中的模型服务于一些简单的网页。为了方便这些网页的构建，该模型使用 `HTMLPageAssembler` 角色。该角色将来自输入端口的内容插入指定的模板文件中，并输出结果 HTML 页面。在这个模板文件中，输入端口的名称与 HTML 标记 ID 相匹配。

例 16.5 图 16-19 展示了图 16-17 中 `HTMLPageAssembler` 角色引用的 HTML 模板。注意具有 ID “welcomeMessage”的 `div` 标记。更加值得注意的是，每个角色都有一个由模型构建器添加的名为 `welcomeMessage` 的输入端口。该端口收到的所有信息都要插入响应 HTML 页面中，`div` 标记位置。

```

1      <body>
2      <div>
3          <div id="welcomeMessage">
4          </div>
5
6          <div> <p> You can: </p> </div>
7          <form accept-charset="UTF-8" action="cookies
8              method="post">
9              <p> Save a cookie with your name </p>
10             <p> Please enter a name:
11                 <input type="text" name="name" id="name"/>
12                 <br>
13                 <input type="submit" value="Save cookie"/>
14             </p>
15             </form>
16
17             <div> <p> Delete the cookie with your name </p>
18                 <input type="button" value="Delete cookie"
19                     onclick="deleteCookie()" />
20             </div>
21
22             <div> <p>
23                 Refresh the page for a personalized greeting
24             </p> </div>
25

```

图 16-19 在图 16-17 中 `HTMLPageAssembler` 角色引用的 HTML 模板


```

26         <form name="input" action="/cookies" method="get">
27             <input type="submit" value="Refresh page" />
28         </form>
29
30     </div>
31 </body>
32 </html>

```

图 16-19 (续)

注意 Save cookie 和 Refresh page 按钮是 HTML 表单。单击这些按钮时，执行指定的操作。例如，Save cookie 按钮向有关的 URL cookies，如 `http://localhost:8078/cookies`，图 16-19 中第 7 行指定，生成一个 POST 请求。Refresh page 按钮向相同的 URL(第 24 行指定的)生成一个 GET 请求。

图 16-17 中的另一种技术是使用 JavaScript 来更新页面，而不是返回一个新页面。这种技术称为 AJAX (异步 JavaScript 和 XML)。

例 16.6 Delete cookie 按钮调用 JavaScript 函数 `deleteCookie()`，如图 16-19 的第 17~18 行所示。图 16-20 显示了 `deleteCookie()` 函数的定义。该函数向相关的 URL `cookies/delete` 提交一个 POST 请求。如果请求成功，那么将响应数据插入具有 ID `welcomeMessage` 的 HTML 元素中(覆盖任何以前的数据)。如果请求不成功，在这个元素中插入一条错误消息。

```

1  <!DOCTYPE HTML>
2  <html>
3      <head>
4          <script type="text/javascript"
5              src="http://code.jquery.com/jquery-1.6.4.min.js">
6          </script>
7          <script type="text/javascript">
8              function deleteCookie() {
9                  jQuery.ajax({
10                      url: "/cookies/delete",
11                      type: "post",
12                      success: function(data) {
13                          jQuery('#welcomeMessage')
14                              .html(data);
15                      },
16                      error: function(data) {
17                          jQuery('#welcomeMessage')
18                              .html("Error deleting cookie.");
19                      }
20                  });
21              }
22          </script>
23          <title>Cookies demo</title>
24
25      </head>

```

图 16-20 在图 16-17 中使用的 HTML 模板页面的头部区

这个例子说明使用 Ajax 的两个原因。首先，对于删除的情况，返回整个页面是没有必要的。一条简单的消息就足够了。在许多情况下开发人员可能要将一个小的更新插入一个更大的页面中，这可以促进关注点分离，这里一个开发人员可以负责主要页面，第二个可以负责更新而无需知道其他主页面的结构，第二个开发人员也可能还想创建一个 Web 服务来为

许多不同的页面提供数据。

使用 Ajax 的更微妙的原因是,网站的 URL 保持不变, `http://localhost:8078/cookies`, 当仍然能够通过一个 URL 结构删除 Web 服务, `cookies/delete` 时。如果 URL 变为 `http://localhost:8078/cookies/delete`, 那么用户在单击 **further** 按钮时就会出现問題, 因为按钮 URL 是作为相关 URL 来定义的。例如, URL 会变为 `http://localhost:8078/cookies/delete/cookies`。

当然,还有许多其他的方法可创建网页以响应 HTTP 请求。一个特别有趣的可能性是使用 16.1 节中所涉及的技术,从 Ptolemy II 模型生成网页。事实上,Web 服务模型可能包括 `HTMLModelExporter` 角色的实例,它引用另一个 Ptolemy II 模型,执行它,生成带有结果的网页,并返回网页。这样就提出了一个特别强大的方式来组合模型以便提供更复杂的服务。

16.3 小结

通过构建模型来创建网页和 Web 服务,是一个非常实用的以模块化方式组合复杂组件的方式。至少,导出包含模型的网页是很有价值的,它使设计者团队之间能更有效地沟通。更有趣的是,将 Web 服务器同模型合并为构建分布式服务提供了一个强有力的方式。

练习

在第 4 章的例 4.3 中讨论的图 4-3,实现了一个简单的聊天客户端,它使用 HTTP Get 和 HTTP Post 在因特网上启用客户端与其他客户聊天。在这个练习中,构建一个简单的(有限的)Web 服务器来支持这个客户端。该服务器将支持两个客户端,一个使用以下 URL

`http://localhost:8078/chat/Claudius`

用于它的 Get 请求,另一个使用以下 URL

`http://localhost:8078/chat/Ptolemaeus`

用于它的 Post 请求。两者将使用相同的 URL

`http://localhost:8078/chat/post`

去传递聊天信息。

- (a) 该服务器的一个关键属性是它必须采用长轮询,这里它将一直发送 HTTP Get 请求直到聊天客户端发出一个 HTTP Post,它提供了一些聊天文本,然后它响应所有用 Post 的内容挂起的客户端。为了支持这个功能,创建一个带有 2 个输入端口 (get 和 post) 和一个输出端口 (response) 的面向角色类型的复合角色。这个类应该对 Get 请求排队(最多的一个),当一个 Post 到达时,如果队列中有挂起的 Get 请求,那么它应该用 Post 的内容响应。
- (b) 使用 (a) 中创建的类来构建一个支持两个客户端的 Web 服务器。
- (c) 图 4-3 中聊天客户端的限制是,它不能优雅地停止。Vergil 窗口的 stop (停止) 按钮最终能使它停止,但直到 `FileReader` 角色超时,它还没有停止,这可能需要很长一段时间。在更好的设计中,服务总是在一些有限的时间内响应 HTTP Get 请求,这些时间由 `maximumResponseTime` 参数给出。它可以使用一个空字符串来响应,并且客户端能够过滤空字符串以避免将它们显示给用户。在设计中,停止客户端将在 `maximumResponseTime` 参数给定的时间内完成。修改服务器和客户端来实现这一点。
- (d) (开放题) 在前面的提问中,要求设计的 Web 服务器只能支持两个客户端。另外,没有客户端的身份验证。讨论如何处理这些不足,并实现一个至少解决其中一个问题的更复杂的 Web 服务。

信号显示

Christopher Brooks 和 Edward A. Lee

Ptolemy II 包含许多信号绘图仪 (signal plotter), 如图 17-1 所示。可以在 Sinks 库中找到这些信号绘图仪, 如图 17-2 所示。附录对这些功能进行了概述, 并着重介绍如何定制功能。定制成功后, 保存包含绘图仪的模型将一直保存这些定制。

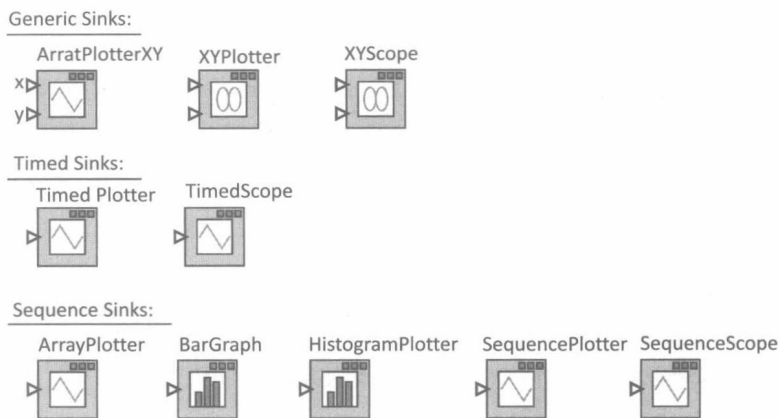


图 17-1 可用信号绘图仪

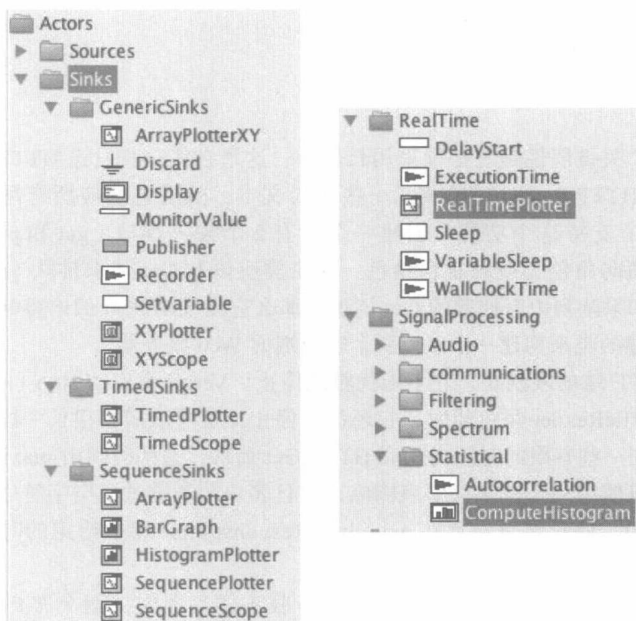


图 17-2 Sinks 库中的信号绘图仪

17.1 可用绘图仪概述

图 17-1 所示的绘图仪提供了许多建立在共同框架上的功能。最基本的是 **Sequence Plotter**，它只简单描绘输入端口接收的数据值，如图 17-3 所示。定制标题、坐标、图例和信号图的机制将在下一节介绍。

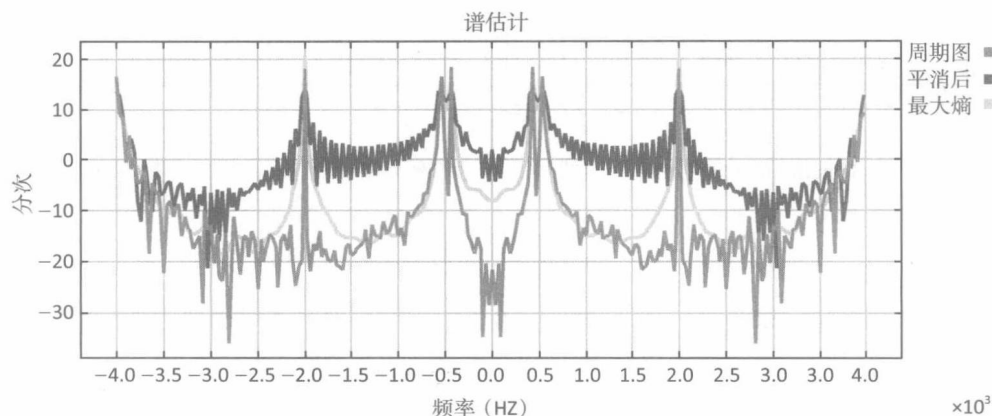


图 17-3 由 SequencePlotter 角色产生的图形的例子

图 17-1 中显示的角色提供多种显示数据的方式。例如 **ArrayPlotter** 对输入数组进行操作而不对序列进行操作。而 **SequencePlotter** 角色绘制模型整个运行期间的所有输入数据，**SequenceScope** 角色绘制输入数据的窗口，并选择性地覆盖它们，如图 17-4 所示。长时间运行时，**SequenceScope** 角色更实用。它工作起来像示波器（oscilloscope）那样不保存旧的数据，只绘制新数据，而且不断覆盖数据的窗口。

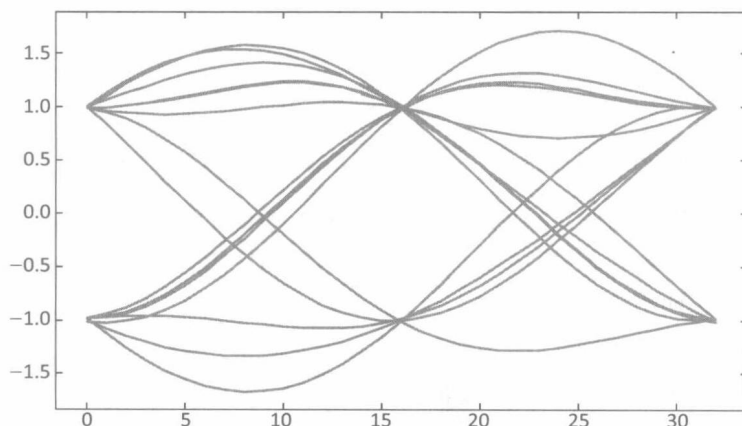


图 17-4 由 SequenceScope 角色产生的图的例子，它覆盖连续的数据窗口，与示波器的工作方式类似

TimedPlotter 角色绘制作输入时间戳函数的输入数据，如图 17-5 所示。在推进模型时间域中（例如，DE 和 Continuous），该绘图仪比较实用。**TimedScope** 与此类似，虽然与 **SequenceScope** 类似，但它工作起来更像示波器那样不保存旧的数据。

XYPlotter 角色绘制一个输入端口的输入数据与另一个输入端口的输入数据，如图 17-6 所示。**XYScope** 与此类似，它虽与 **SequenceScope** 类似，但它工作起来像示波器那样不保存旧的数据。**ArrayPlotterXY** 也一样，除了它是对数组而不是对序列进行操作。

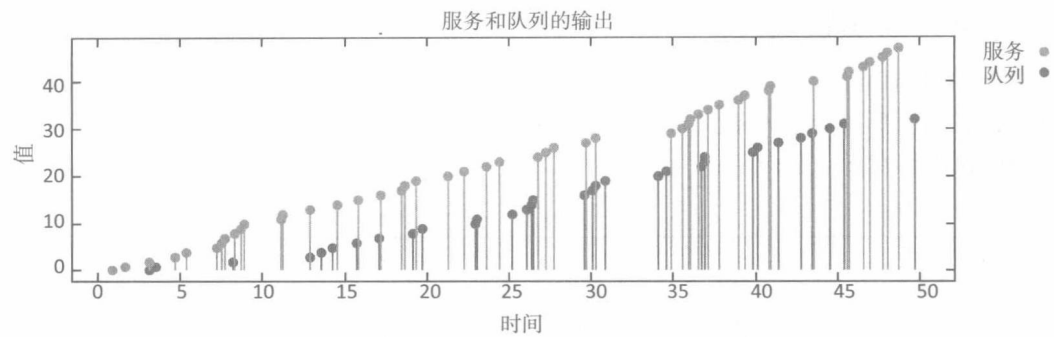


图 17-5 由 TimedPlotter 角色产生的图的例子，它将输入数据作为输入的时间戳的函数绘制

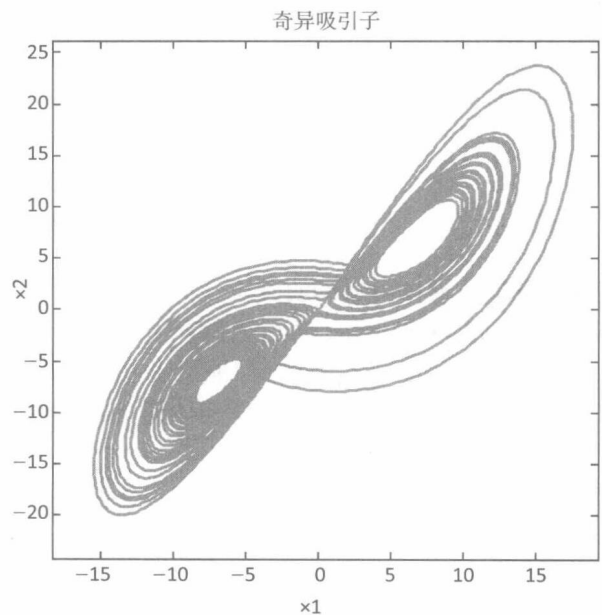


图 17-6 由 XYPlotter 角色产生的图的例子，它绘制一个输入端口的输入数据在与其他端口的输入数据

BarGraph 以柱状图的形式绘制输入数组。HistogramPlotter 计算输入数据的直方图，然后以柱状图的方式绘制，如图 17-7 所示。在图 17-2 中，还有一个 ComputeHistogram 角色，它只计算直方图而不绘制它。

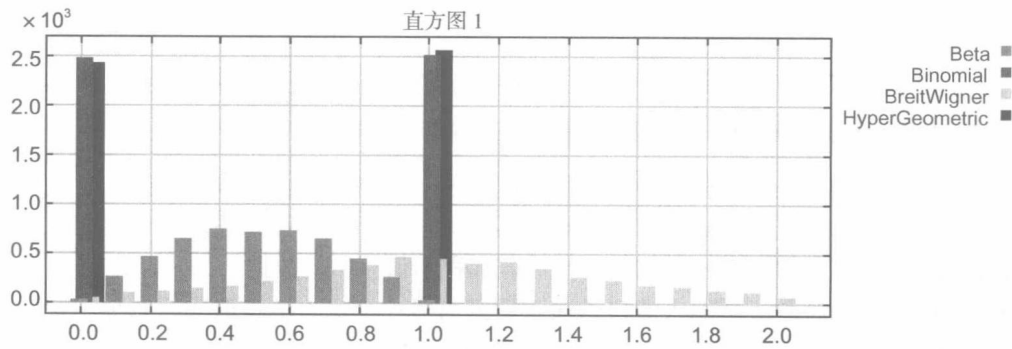


图 17-7 HistogramPlotter 角色参数的图的例子，它计算和绘制基于输入数据的直方图

如图 17-2 所示, RealTimePlotter 将在计算机执行模型时实时显示输入结果。

17.2 绘图仪定制

默认情况下, SequencePlotter 输出如图 17-8 和图 3.1 所示。默认的标题不提供具体信息, 轴没有刻度, 水平轴的范围从 0 ~ 255, 没有任何含义[⊖]。在模型中创建绘图仪, 如图 3-1 所示, 在一次迭代中, Spectrum 角色产生 256 个输出令牌。默认情况下, SequencePlotter 把这些样本从 0 ~ 255 计数, 并将这些数字作为图像的水平轴。但在模型中, 水平轴应具有更多的含义。在这个特殊的例子中, 绘制的数据代表频率点, 它们的变化范围为 $-\pi \sim \pi$ 的弧度 / 每秒。

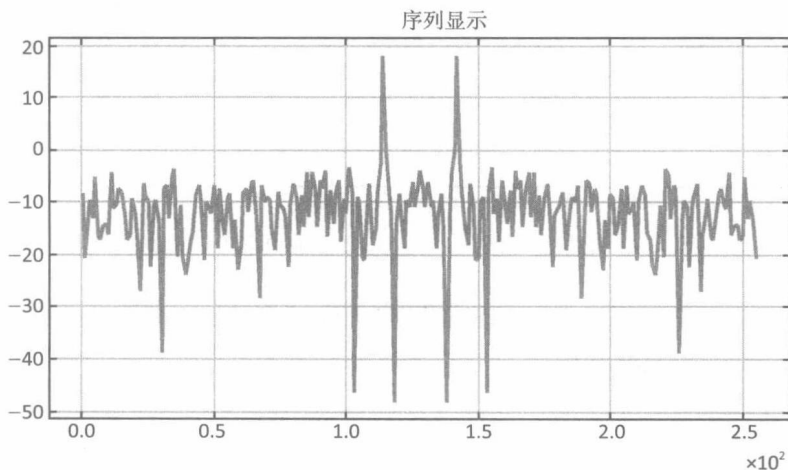


图 17-8 默认情况下, SequencePlotter 角色产生的图没有信息标签

SequencePlotter 角色有一些相关参数, 如图 17-9 所示, 这些可以用来改善图形的标号。xInit 参数为第一个令牌指定水平轴上要使用的值。xUnit 参数为每一个后续令牌指定递增的值。分别将这两个参数设置为 “ $-\pi$ ” 和 “ $\pi/128$ ”, 产生的图如图 17-10 所示。

fillOnWrapup:	<input checked="" type="checkbox"/>
automaticRescale:	<input type="checkbox"/>
legend:	
startingDataset:	0
xInit:	$-\pi$
xUnit:	$\pi/128$
firingsPerIteration:	256

图 17-9 SequencePlotter 角色的参数

尽管该图很完善, 但还是丢失了一些有用的信息。为了更精确地控制绘图仪的可视化参数, 单击图的右上角按钮行从右数第二个按钮。这个按钮能启动一个格式控制窗口, 如图

⊖ 暗示: 需要注意右下有一个 “x102”, 这表明标签 “2.5” 代表 “250”。

17-11 所示，键入数值后得到图 17-12 所示的图。其中大部分都有注释，下面说明一些典型的用法：

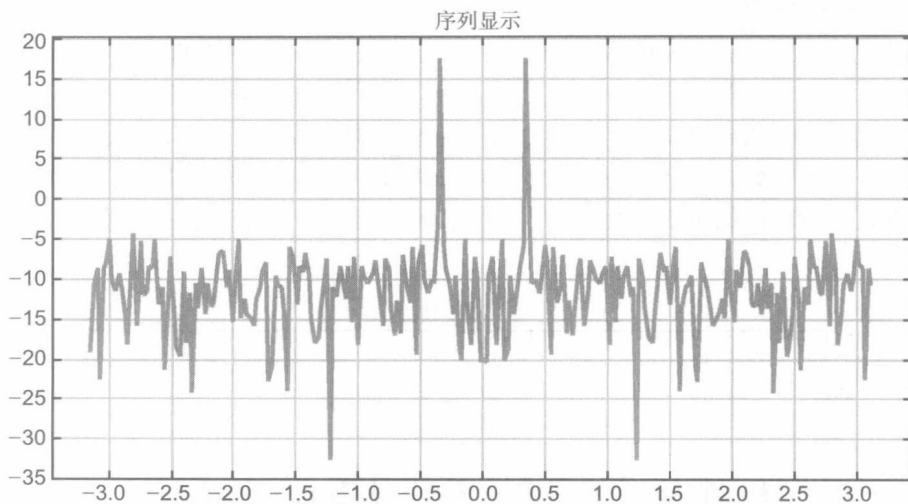


图 17-10 更好的带标记的图，其中水平轴能更准确地代表频率值

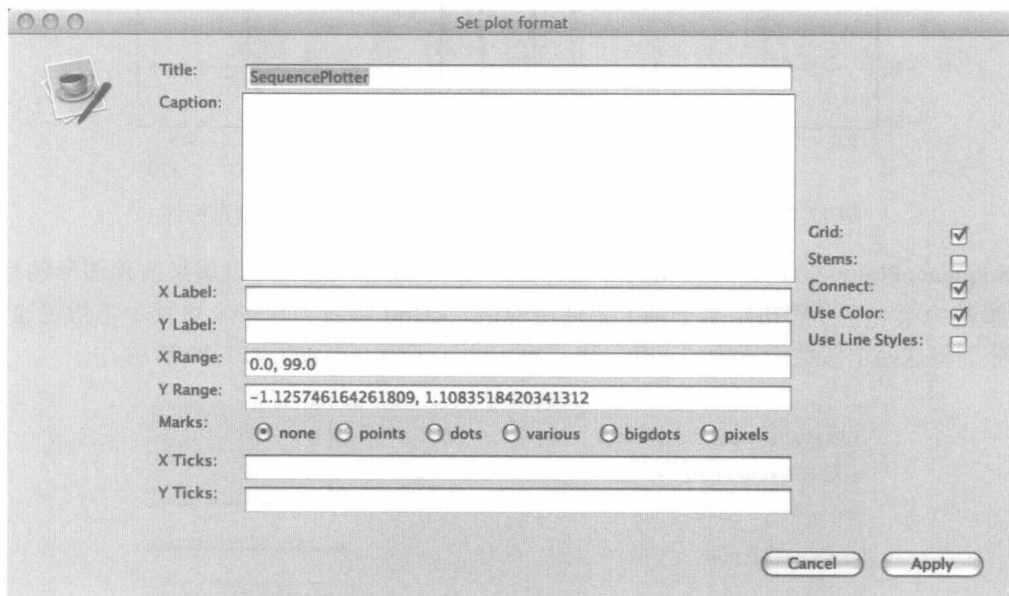


图 17-11 图的格式控制面板

- 关闭 Grid（网格）可避免显示混乱。
- 添加标题和轴标签。
- X 和 Y 的范围取决于图形右上角的填充按钮。
- Stem 图可以通过单击“Stems”显示。
- 单个令牌可以通过单击“dots”显示。
- 连接线路可以通过取消勾选“连接”被消除。
- X 轴刻度改成了 $\pi/2$ 的倍数。这是通过输入以下内容到 X Ticks 字数来实现的：

-PI -3.14159, -PI/2 -1.570795, 0 0.0, PI/2 1.570795, PI 3.14159

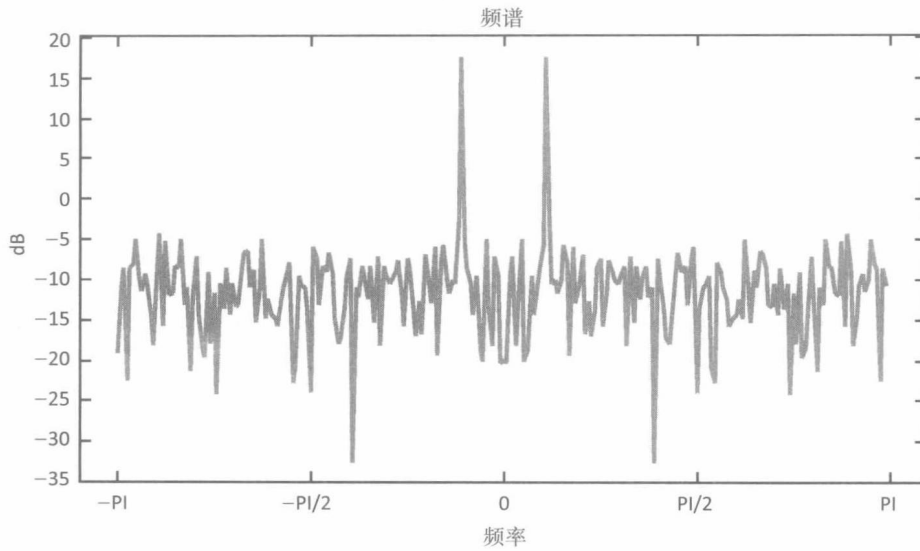


图 17-12 还是更好标记的图

一般的语法是: label, value, label, value, ..., 其中的标签可以是任何字符串 (如果有空格则加引号), 而值则为数字。

参考文献

- Agha, G. A., I. A. Mason, S. F. Smith, and C. L. Talcott, 1997: A foundation for actor computation. *Journal of Functional Programming*, **7**(1), 1–72.
- Allen, F. E., 1970: Control flow analysis. *SIGPLAN Notices*, **5**(7), 1–19.
- Alur, R., S. Kannan, and M. Yannakakis, 1999: Communicating hierarchical state machines. In *26th International Colloquium on Automata, Languages, and Programming*, Springer, vol. LNCS 1644, pp. 169–178.
- Andalam, S., P. S. Roop, and A. Girault, 2010: Predictable multithreading of embedded applications using PRET-C. In *Formal Methods and Models for Codesign (MEMOCODE)*, IEEE/ACM, Grenoble, France, pp. 159–168. doi:10.1109/MEMCOD.2010.5558636.
- André, C., 1996: SyncCharts: a visual representation of reactive behaviors. Tech. Rep. RR 95–52, revision: RR (96–56), University of Sophia-Antipolis. Available from: <http://www-sop.inria.fr/members/Charles.Andre/CA%20Publis/SYNCCHARTS/overview.html>.
- André, C., F. Mallet, and R. d. Simone, 2007: Modeling time(s). In *Model Driven Engineering Languages and Systems (MoDELS/UML)*, Springer, Nashville, TN, vol. LNCS 4735, pp. 559–573. doi:10.1007/978-3-540-75209-7_38.
- Arbab, F., 2006: A behavioral model for composition of software components. *L'Object, Lavoisier*, **12**(1), 33–76. doi:10.3166/objet.12.1.33-76.
- Arvind, L. Bic, and T. Ungerer, 1991: Evolution of data-flow computers. In Gaudiot, J.-L. and L. Bic, eds., *Advanced Topics in Data-Flow Computing*, Prentice-Hall.
- Baccelli, F., G. Cohen, G. J. Olster, and J. P. Quadrat, 1992: *Synchronization and Linearity, An Algebra for Discrete Event Systems*. Wiley, New York.
- Baier, C. and M. E. Majster-Cederbaum, 1994: Denotational semantics in the CPO and metric approach. *Theoretical Computer Science*, **135**(2), 171–220.
- Balarin, F., H. Hsieh, L. Lavagno, C. Passerone, A. L. Sangiovanni-Vincentelli, and Y. Watanabe, 2003: Metropolis: an integrated electronic system design environment. *Computer*, **36**(4).
- Baldwin, P., S. Kohli, E. A. Lee, X. Liu, and Y. Zhao, 2004: Modeling of sensor nets in Ptolemy II. In *Information Processing in Sensor Networks (IPSN)*, Berkeley, CA, USA. Available from: <http://ptolemy.eecs.berkeley.edu/publications/papers/04/VisualSense/>.
- , 2005: Visualsense: Visual modeling for wireless and sensor network systems. Technical Report UCB/ERL M05/25, EECS Department, University of California. Available from: <http://ptolemy.eecs.berkeley.edu/publications/papers/05/visualsense/index.htm>.

- Basu, A., M. Bozga, and J. Sifakis, 2006: Modeling heterogeneous real-time components in BIP. In *International Conference on Software Engineering and Formal Methods (SEFM)*, Pune, pp. 3–12.
- Benveniste, A. and G. Berry, 1991: The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, **79(9)**, 1270–1282.
- Benveniste, A., P. Caspi, P. Le Guernic, and N. Halbwachs, 1994: Data-flow synchronous languages. In Bakker, J. W. d., W.-P. d. Roever, and G. Rozenberg, eds., *A Decade of Concurrency Reflections and Perspectives*, Springer-Verlag, Berlin, vol. 803 of *LNCS*, pp. 1–45.
- Benveniste, A. and P. Le Guernic, 1990: Hybrid dynamical systems theory and the SIGNAL language. *IEEE Tr. on Automatic Control*, **35(5)**, 525–546.
- Berry, G., 1976: Bottom-up computation of recursive programs. *Revue Franais d'Automatique, Informatique et Recherche Oprationnelle*, **10(3)**, 47–82.
- , 1999: *The Constructive Semantics of Pure Esterel - Draft Version 3*. Book Draft. Available from: <http://www-sop.inria.fr/meije/esterel/doc/main-papers.html>.
- , 2003: The effectiveness of synchronous languages for the development of safety-critical systems. White paper, Esterel Technologies. Available from: <http://www.esterel-technologies.com>.
- Berry, G. and G. Gonthier, 1992: The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, **19(2)**, 87–152. Available from: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.17.5606>.
- Bhattacharya, B. and S. S. Bhattacharyya, 2000: Parameterized dataflow modeling of DSP systems. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Istanbul, Turkey, pp. 1948–1951.
- Bhattacharyya, S. S., J. T. Buck, S. Ha, and E. Lee, 1995: Generating compact code from dataflow specifications of multirate signal processing algorithms. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, **42(3)**, 138–150. doi:10.1109/81.376876.
- Bhattacharyya, S. S., J. T. Buck, S. Ha, and E. A. Lee, 1993: A scheduling framework for minimizing memory requirements of multirate DSP systems represented as dataflow graphs. In *VLSI Signal Processing VI*, IEEE, Veldhoven, The Netherlands, pp. 188–196. doi:10.1109/VLSISP.1993.404488.
- Bhattacharyya, S. S. and E. A. Lee, 1993: Scheduling synchronous dataflow graphs for efficient looping. *Journal of VLSI Signal Processing Systems*, **6(3)**, 271–288. doi:10.1007/BF01608539.
- Bhattacharyya, S. S., P. Murthy, and E. A. Lee, 1996a: APGAN and RPMC: Complementary heuristics for translating DSP block diagrams into efficient software implementations. *Journal of Design Automation for Embedded Systems*, **2(1)**, 33–60. doi:10.1023/A:1008806425898.

- Bhattacharyya, S. S., P. K. Murthy, and E. A. Lee, 1996b: *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, Norwell, Mass.
- Bilsen, G., M. Engels, R. Lauwereins, and J. A. Peperstraete, 1996: Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, **44**(2), 397–408. doi:10.1109/78.485935.
- Bock, C., 2006: SysML and UML 2 support for activity modeling. *Systems Engineering*, **9**(2), 160–185.
- Booch, G., I. Jacobson, and J. Rumbaugh, 1998: *The Unified Modeling Language User Guide*. Addison-Wesley.
- Boussinot, F., 1991: Reactive c: An extension to c to program reactive systems. *Software Practice and Experience*, **21**(4), 401–428.
- Box, G. E. P. and N. R. Draper, 1987: *Empirical Model-Building and Response Surfaces*. Wiley Series in Probability and Statistics, Wiley.
- Brock, J. D. and W. B. Ackerman, 1981: Scenarios, a model of non-determinate computation. In *Conference on Formal Definition of Programming Concepts*, Springer-Verlag, vol. LNCS 107, pp. 252–259.
- Broenink, J. F., 1997: Modelling, simulation and analysis with 20-Sim. *CACSD*, **38**(3), 22–25.
- Brooks, C., C. Cheng, T. H. Feng, E. A. Lee, and R. von Hanxleden, 2008: Model engineering using multimodeling. In *International Workshop on Model Co-Evolution and Consistency Management (MCCM)*, Toulouse, France. Available from: <http://chess.eecs.berkeley.edu/pubs/486.html>.
- Brooks, C., E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng, 2004: Heterogeneous concurrent modeling and design in Java. Tech. Rep. Technical Memorandum UCB/ERL M04/16, University of California. Available from: <http://ptolemy.eecs.berkeley.edu/papers/04/ptIIDesignSoftware/>.
- Brooks, C. H. and E. A. Lee, 2003: Ptolemy II coding style. Tech. Rep. Technical Memorandum UCB/ERL M03/44, University of California at Berkeley. Available from: <http://ptolemy.eecs.berkeley.edu/publications/papers/03/codingstyle/>.
- Broy, M., 1983: Applicative real time programming. In *Information Processing 83, IFIP World Congress*, North Holland Publ. Company, Paris, pp. 259–264.
- Broy, M. and G. Stefanescu, 2001: The algebra of stream processing functions. *Theoretical Computer Science*, **258**, 99–129.
- Bryant, V., 1985: *Metric Spaces - Iteration and Application*. Cambridge University Press.
- Buck, J. T., 1993: Scheduling dynamic dataflow graphs with bounded memory using the token flow model. Ph.D. Thesis Tech. Report UCB/ERL 93/69, University of California, Berkeley. Available from: <http://ptolemy.eecs.berkeley.edu/publications/papers/93/jbuckThesis/>.

- Buck, J. T., S. Ha, E. A. Lee, and D. G. Messerschmitt, 1994: Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. Journal of Computer Simulation, special issue on "Simulation Software Development"*, **4**, 155–182. Available from: <http://ptolemy.eecs.berkeley.edu/publications/papers/94/JEurSim/>.
- Burch, J. R., R. Passerone, and A. L. Sangiovanni-Vincentelli, 2001: Overcoming heterophobia: Modeling concurrency in heterogeneous systems. In *International Conference on Application of Concurrency to System Design*, p. 13.
- Buss, A. H. and P. J. Sanchez, 2002: Building complex models with LEGOs (listener event graph objects). *Winter Simulation Conference (WSC 02)*, **1**, 732–737.
- Cardelli, L., 1997: Type systems. In Tucker, A. B., ed., *The Computer Science and Engineering Handbook*, CRC Press, chap. 103, pp. 2208–2236, <http://lucacardelli.name/Papers/TypeSystems>
- Cardelli, L. and P. Wegner, 1985: On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, **17**(4), 471 – 523.
- Carlioni, L. P., R. Passerone, A. Pinto, and A. Sangiovanni-Vincentelli, 2006: Languages and tools for hybrid systems design. *Foundations and Trends in Electronic Design Automation*, **1**(1/2). doi:10.1561/10000000001.
- Caspi, P., P. Raymond, and S. Tripakis, 2007: Synchronous Programming. In Lee, I., J. Leung, and S. Son, eds., *Handbook of Real-Time and Embedded Systems*, Chapman & Hall, pp. 14–1 — 14–21. Available from: <http://www-verimag.imag.fr/~tripakis/papers/handbook07.pdf>.
- Cassandras, C. G., 1993: *Discrete Event Systems, Modeling and Performance Analysis*. Irwin.
- Cataldo, A., E. A. Lee, X. Liu, E. Matsikoudis, and H. Zheng, 2006: A constructive fixed-point theorem and the feedback semantics of timed systems. In *Workshop on Discrete Event Systems (WODES)*, Ann Arbor, Michigan. Available from: <http://ptolemy.eecs.berkeley.edu/publications/papers/06/constructive/>.
- Chandy, K. M. and J. Misra, 1979: Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. on Software Engineering*, **5**(5), 440–452.
- Clarke, E. M., O. Grumberg, and D. A. Peled, 2000: *Model checking*. MIT Press, ISBN 0-262-03270-8.
- Coffman, E. G., Jr. (Ed), 1976: *Computer and Job Scheduling Theory*. Wiley.
- Conway, M. E., 1963: Design of a separable transition-diagram compiler. *Communications of the ACM*, **6**(7), 396–408.
- Corbett, J. C., J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak,

- C. Taylor, R. Wang, and D. Woodford, 2012: Spanner: Googles globally-distributed database. In *OSDI*.
- Cousot, P. and R. Cousot, 1977: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages (POPL)*, ACM Press, pp. 238–252.
- Creeger, M., 2005: Multicore CPUs for the masses. *ACM Queue*, **3(7)**, 63–64.
- Davey, B. A. and H. A. Priestly, 2002: *Introduction to Lattices and Order*. Cambridge University Press, second edition ed.
- de Alfaro, L. and T. Henzinger, 2001: Interface automata. In *ESEC/FSE 01: the Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*.
- Dennis, J. B., 1974: First version data flow procedure language. Tech. Rep. MAC TM61, MIT Laboratory for Computer Science.
- Derler, P., E. A. Lee, and S. Matic, 2008: Simulation and implementation of the ptides programming model. In *IEEE International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, Vancouver, Canada.
- Dijkstra, E. W., 1968: Go to statement considered harmful (letter to the editor). *Communications of the ACM*, **11(3)**, 147–148.
- Edwards, S. A. and E. A. Lee, 2003a: The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming*, **48(1)**, 21–42. doi: 10.1016/S0167-6423(02)00096-5.
- , 2003b: The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming*, **48(1)**, 21–42. Available from: <http://ptolemy.eecs.berkeley.edu/papers/03/blockdiagram/>.
- Eidson, J. C., 2006: *Measurement, Control, and Communication Using IEEE 1588*. Springer. doi:10.1007/1-84628-251-9.
- Eidson, J. C., E. A. Lee, S. Matic, S. A. Seshia, and J. Zou, 2012: Distributed real-time software for cyber-physical systems. *Proceedings of the IEEE (special issue on CPS)*, **100(1)**, 45–59. doi:10.1109/JPROC.2011.2161237.
- Eker, J. and J. W. Janneck, 2003: Cal language report: Specification of the cal actor language. Tech. Rep. Technical Memorandum No. UCB/ERL M03/48, University of California, Berkeley, CA. Available from: <http://ptolemy.eecs.berkeley.edu/papers/03/Cal/index.htm>.
- Eker, J., J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, 2003: Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, **91(2)**, 127–144. Available from: <http://www.ptolemy.eecs.berkeley.edu/publications/papers/03/TamingHeterogeneity/>.
- Encyclopedia Britannica, 2010: Ockham’s razor. *Encyclopedia Britannica Online*,

- Retrieved June 24, 2010. Available from: <http://www.britannica.com/EBchecked/topic/424706/Ockhams-razor>.
- Falk, J., J. Keiner, C. Haubelt, J. Teich, and S. S. Bhattacharyya, 2008: A generalized static data flow clustering algorithm for mp soc scheduling of multimedia applications. In *Embedded Software (EMSOFT)*, ACM, Atlanta, Georgia, USA.
- Faustini, A. A., 1982: An operational semantics for pure dataflow. In *Proceedings of the 9th Colloquium on Automata, Languages and Programming (ICALP)*, Springer-Verlag, vol. Lecture Notes in Computer Science (LNCS) Vol. 140, pp. 212–224.
- Feng, T. H., 2009: Model transformation with hierarchical discrete-event control. PhD Thesis UCB/EECS-2009-77, EECS Department, UC Berkeley. Available from: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-77.html>.
- Feng, T. H. and E. A. Lee, 2008: Real-time distributed discrete-event execution with fault tolerance. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, IEEE, St. Louis, MO, USA. Available from: <http://chess.eecs.berkeley.edu/pubs/389.html>.
- Feng, T. H., E. A. Lee, H. D. Patel, and J. Zou, 2008: Toward an effective execution policy for distributed real-time embedded systems. In *14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, St. Louis, MO, USA. Available from: <https://chess.eecs.berkeley.edu/pubs/402>.
- Feng, T. H., E. A. Lee, and L. W. Schruben, 2010: Ptera: An event-oriented model of computation for heterogeneous systems. In *EMSOFT*, ACM Press, Scottsdale, Arizona, USA. doi:10.1145/1879021.1879050.
- Feredj, M., F. Boulanger, and A. M. Mbobi, 2009: A model of domain-polymorph component for heterogeneous system design. *The Journal of Systems and Software*, **82**, 112–120.
- Fitzgerald, J., P. G. Larsen, K. Pierce, M. Verhoef, and S. Wolff, 2010: Collaborative modelling and co-simulation in the development of dependable embedded systems. In *Integrated Formal Methods (IFM)*, Springer-Verlag, vol. LNCS 6396, pp. 12–26. doi:10.1007/978-3-642-16265-7_2.
- Fitzgerald, J. S., P. G. Larsen, and M. Verhoef, 2008: Vienna development method. In *Wiley Encyclopedia of Computer Science and Engineering*, John Wiley & Sons, Inc. doi:10.1002/9780470050118.ecse447.
- Foley, J., A. van Dam, S. Feiner, and J. Hughes, 1996: *Computer Graphics, Principles and Practice*. Addison-Wesley, 2nd ed.
- Friedman, D. P. and D. S. Wise, 1976: CONS should not evaluate its arguments. In *Third Int. Colloquium on Automata, Languages, and Programming*, Edinburg University Press.
- Fritzson, P., 2003: *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley.
- Fuhrmann, H. and R. v. Hanxleden, 2010: Taming graphical modeling. In *Model Driven Engineering Languages and Systems (MODELS) 13th International Confer-*

- ence, *MODELS 2010, Oslo, Norway, October 3-8, 2010*, Springer, Oslo, Norway, vol. 6394, pp. 196–210. doi:10.1007/978-3-642-16145-2_14.
- Fuhrmann, H. and R. von Hanxleden, 2008: On the pragmatics of model-based design. In *Foundations of Computer Software. Future Trends and Techniques for Development – Monterey Workshop*, Springer, Budapest, Hungary, vol. LNCS 6028, pp. 116–140. doi:10.1007/978-3-642-12566-9_7.
- Fujimoto, R., 2000: *Parallel and Distributed Simulation Systems*. John Wiley and Sons.
- Gaderer, G., P. Loschmidt, E. G. Cota, J. H. Lewis, J. Serrano, M. Cattin, P. Alvarez, P. M. Oliveira Fernandes Moreira, T. Wlostowski, J. Dedic, C. Prados, M. Kreider, R. Baer, S. Rauch, and T. Fleck, 2009: The white rabbit project. In *Int. Conf. on Accelerator and Large Experimental Physics Control Systems*, Kobe, Japan.
- Galletly, J., 1996: *Occam-2*. University College London Press, 2nd ed.
- Ganter, B. and R. Wille, 1998: *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag, Berlin.
- Geilen, M. and T. Basten, 2003: Requirements on the execution of Kahn process networks. In *European Symposium on Programming Languages and Systems*, Springer, LNCS, pp. 319–334. Available from: <http://www.ics.ele.tue.nl/~tbasten/papers/esop03.pdf>.
- Geilen, M., T. Basten, and S. Stuijk, 2005: Minimising buffer requirements of synchronous dataflow graphs with model checking. In *Design Automation Conference (DAC)*, ACM, Anaheim, California, USA, pp. 819–824. doi:10.1145/1065579.1065796.
- Geilen, M. and S. Stuijk, 2010: Worst-case performance analysis of synchronous dataflow scenarios. In *CODES+ISSS*, ACM, Scottsdale, Arizona, USA, pp. 125–134.
- Girault, A., B. Lee, and E. A. Lee, 1999: Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions On Computer-aided Design Of Integrated Circuits And Systems*, **18(6)**, 742–760.
- Goderis, A., C. Brooks, I. Altintas, E. A. Lee, and C. Goble, 2009: Heterogeneous composition of models of computation. *Future Generation Computer Systems*, **25(5)**, 552–560. doi:doi:10.1016/j.future.2008.06.014.
- Goessler, G. and A. Sangiovanni-Vincentelli, 2002: Compositional modeling in Metropolis. In *Second International Workshop on Embedded Software (EMSOFT)*, Springer-Verlag, Grenoble, France.
- Golomb, S. W., 1971: Mathematical models: Uses and limitations. *IEEE Transactions on Reliability*, **R-20(3)**, 130–131. doi:10.1109/TR.1971.5216113.
- Gu, Z., S. Wang, S. Kodase, and K. G. Shin, 2003: An end-to-end tool chain for multi-view modeling and analysis of avionics mission computing software. In *Real-Time Systems Symposium (RTSS)*, pp. 78 – 81.
- Ha, S. and E. A. Lee, 1991: Compile-time scheduling and assignment of dataflow program graphs with data-dependent iteration. *IEEE Transactions on Computers*, **40(11)**, 1225–1238. doi:10.1109/12.102826.

- Halbwachs, N., P. Caspi, P. Raymond, and D. Pilaud, 1991: The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, **79(9)**, 1305–1319.
- Hardebolle, C. and F. Boulanger, 2007: ModHel'X: A component-oriented approach to multi-formalism modeling. In *MODELS 2007 Workshop on Multi-Paradigm Modeling*, Elsevier Science B.V., Nashville, Tennessee, USA.
- Harel, D., 1987: Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, **8(3)**, 231–274.
- Harel, D., H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, 1990: STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, **16(4)**, 403–414. doi:10.1109/32.54292.
- Harel, D. and A. Pnueli, 1985: On the development of reactive systems. In Apt, K. R., ed., *Logic and Models for Verification and Specification of Concurrent Systems*, Springer-Verlag, vol. F13 of *NATO ASI Series*, pp. 477–498.
- Henzinger, T. A., 2000: The theory of hybrid automata. In Inan, M. and R. Kurshan, eds., *Verification of Digital and Hybrid Systems*, Springer-Verlag, vol. 170 of *NATO ASI Series F: Computer and Systems Sciences*, pp. 265–292.
- Henzinger, T. A., B. Horowitz, and C. M. Kirsch, 2001: Giotto: A time-triggered language for embedded programming. In *EMSOFT 2001*, Springer-Verlag, Tahoe City, CA, vol. LNCS 2211, pp. 166–184.
- Herrera, F. and E. Villar, 2006: A framework for embedded system specification under different models of computation in SystemC. In *Design Automation Conference (DAC)*, ACM, San Francisco.
- Hewitt, C., 1977: Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, **8(3)**, 323–363.
- Hoare, C. A. R., 1978: Communicating sequential processes. *Communications of the ACM*, **21(8)**, 666–677.
- Hopcroft, J. and J. Ullman, 1979: *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA.
- Hsu, C.-J., F. Keceli, M.-Y. Ko, S. Shahparnia, and S. S. Bhattacharyya, 2004: DIF: An interchange format for dataflow-based design tools. In *International Workshop on Systems, Architectures, Modeling, and Simulation*, Samos, Greece.
- Hu, T. C., 1961: Parallel sequencing and assembly line problems. *Operations Research*, **9(6)**, 841–848.
- Ingalls, R. G., D. J. Morrice, and A. B. Whinston, 1996: Eliminating canceling edges from the simulation graph model methodology. In *WSC '96: Proceedings of the 28th conference on Winter simulation*, IEEE Computer Society, Washington, DC, USA, ISBN 0-7803-3383-7, pp. 825–832.
- Jantsch, A., 2003: *Modeling Embedded Systems and SoCs - Concurrency and Time in Models of Computation*. Morgan Kaufmann.

- Jantsch, A. and I. Sander, 2005: Models of computation and languages for embedded system design. *IEE Proceedings on Computers and Digital Techniques*, **152(2)**, 114–129.
- Jefferson, D., 1985: Virtual time. *ACM Trans. Programming Languages and Systems*, **7(3)**, 404–425.
- Johannessen, S., 2004: Time synchronization in a local area network. *IEEE Control Systems Magazine*, 61–69.
- Johnston, W. M., J. R. P. Hanna, and R. J. Millar, 2004: Advances in dataflow programming languages. *ACM Computing Surveys*, **36(1)**, 1–34.
- Kahn, G., 1974: The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*, North-Holland Publishing Co., pp. 471–475.
- Kahn, G. and D. B. MacQueen, 1977: Coroutines and networks of parallel processes. In Gilchrist, B., ed., *Information Processing*, North-Holland Publishing Co., pp. 993–998.
- Karsai, G., A. Lang, and S. Neema, 2005: Design patterns for open tool integration. *Software and Systems Modeling*, **4(2)**, 157–170. doi:10.1007/s10270-004-0073-y.
- Kay, S. M., 1988: *Modern Spectral Estimation: Theory & Application*. Prentice-Hall, Englewood Cliffs, NJ.
- Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, 1997: Aspect-oriented programming. In *ECOOP, European Conference in Object-Oriented Programming*, Springer-Verlag, Finland, vol. LNCS 1241.
- Kienhuis, B., E. Deprettere, P. van der Wolf, and K. Vissers, 2001: A methodology to design programmable embedded systems. In Deprettere, E., J. Teich, and S. Vassiliadis, eds., *Systems, Architectures, Modeling, and Simulation (SAMOS)*, Springer-Verlag, vol. LNCS 2268.
- Kodosky, J., J. MacCriskin, and G. Rymar, 1991: Visual programming using structured data flow. In *IEEE Workshop on Visual Languages*, IEEE Computer Society Press, Kobe, Japan, pp. 34–39.
- Kopetz, H., 1997: *Real-Time Systems : Design Principles for Distributed Embedded Applications*. Springer.
- Kopetz, H. and G. Bauer, 2003: The time-triggered architecture. *Proceedings of the IEEE*, **91(1)**, 112–126.
- Lamport, L., R. Shostak, and M. Pease, 1978: Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, **21(7)**, 558–565.
- Landin, P. J., 1965: A correspondence between Algol 60 and Church's lambda notation. *Communications of the ACM*, **8(2)**, 89–101.
- Le Guernic, P., T. Gauthier, M. Le Borgne, and C. Le Maire, 1991: Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, **79(9)**, 1321 – 1336. doi: 10.1109/5.97301.

- Lee, E. A., 1986: A coupled hardware and software architecture for programmable digital signal processors. PhD Thesis UCB/ERL M86/54, University of California. Available from: <http://ptolemy.eecs.berkeley.edu/publications/papers/86/LeePhDThesis/>.
- , 1999: Modeling concurrent real-time processes using discrete events. *Annals of Software Engineering*, **7**, 25–45. doi:10.1023/A:1018998524196.
- , 2006: The problem with threads. *Computer*, **39(5)**, 33–42. doi:10.1109/MC.2006.180.
- , 2008a: Cyber physical systems: Design challenges. In *International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, IEEE, Orlando, Florida, pp. 363–369. doi:10.1109/ISORC.2008.25.
- , 2008b: ThreadedComposite: A mechanism for building concurrent and parallel Ptolemy II models. Technical Report UCB/EECS-2008-151, EECS Department, University of California, Berkeley. Available from: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-151.html>.
- , 2009: Finite state machines and modal models in Ptolemy II. Tech. Rep. UCB/EECS-2009-151, EECS Department, University of California, Berkeley. Available from: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-151.html>.
- , 2010a: CPS foundations. In *Design Automation Conference (DAC)*, ACM, Anaheim, California, USA, pp. 737–742. doi:10.1145/1837274.1837462.
- , 2010b: Disciplined heterogeneous modeling. In Petriu, D. C., N. Rouquette, and O. Haugen, eds., *Model Driven Engineering, Languages, and Systems (MODELS)*, IEEE, pp. 273–287. Available from: <http://chess.eecs.berkeley.edu/pubs/679.html>.
- Lee, E. A., E. Goei, H. Heine, and W. Ho, 1989: Gabriel: A design environment for programmable DSPs. In *Design Automation Conference (DAC)*, Las Vegas, NV, pp. 141–146. Available from: <http://ptolemy.eecs.berkeley.edu/publications/papers/89/gabriel/>.
- Lee, E. A. and S. Ha, 1989: Scheduling strategies for multiprocessor real-time DSP. In *Global Telecommunications Conference (GLOBECOM)*, vol. 2, pp. 1279–1283. doi:10.1109/GLOCOM.1989.64160.
- Lee, E. A., X. Liu, and S. Neuendorffer, 2009a: Classes and inheritance in actor-oriented design. *ACM Transactions on Embedded Computing Systems (TECS)*, **8(4)**, 29:1–29:26. doi:10.1145/1550987.1550992.
- Lee, E. A., S. Matic, S. A. Seshia, and J. Zou, 2009b: The case for timing-centric distributed software. In *IEEE International Conference on Distributed Computing Systems Workshops: Workshop on Cyber-Physical Systems*, IEEE, Montreal, Canada, pp. 57–64. Available from: <http://chess.eecs.berkeley.edu/pubs/607.html>.
- Lee, E. A. and E. Matsikoudis, 2009: The semantics of dataflow with firing. In Huet, G., G. Plotkin, J.-J. Lévy, and Y. Bertot, eds., *From Semantics to Computer Science: Essays in memory of Gilles Kahn*, Cambridge University Press. Available

- from: <http://ptolemy.eecs.berkeley.edu/publications/papers/08/DataflowWithFiring/>.
- Lee, E. A. and D. G. Messerschmitt, 1987a: Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, **C-36(1)**, 24–35. doi:10.1109/TC.1987.5009446.
- , 1987b: Synchronous data flow. *Proceedings of the IEEE*, **75(9)**, 1235–1245. doi:10.1109/PROC.1987.13876.
- Lee, E. A. and S. Neuendorffer, 2000: MoML - a modeling markup language in XML. Tech. Rep. UCB/ERL M00/12, UC Berkeley. Available from: <http://ptolemy.eecs.berkeley.edu/publications/papers/00/moml/>.
- Lee, E. A., S. Neuendorffer, and M. J. Wirthlin, 2003: Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers*, **12(3)**, 231–260. Available from: <http://ptolemy.eecs.berkeley.edu/papers/03/actorOrientedDesign/>.
- Lee, E. A. and T. M. Parks, 1995: Dataflow process networks. *Proceedings of the IEEE*, **83(5)**, 773–801. doi:10.1109/5.381846.
- Lee, E. A. and A. Sangiovanni-Vincentelli, 1998: A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, **17(12)**, 1217–1229. Available from: <http://ptolemy.eecs.berkeley.edu/publications/papers/98/framework/>.
- Lee, E. A. and S. A. Seshia, 2011: *Introduction to Embedded Systems - A Cyber-Physical Systems Approach*. LeeSeshia.org, Berkeley, CA. Available from: <http://LeeSeshia.org>.
- Lee, E. A. and S. Tripakis, 2010: Modal models in Ptolemy. In *3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT)*, Linköping University Electronic Press, Linköping University, Oslo, Norway, vol. 47, pp. 11–21. Available from: <http://chess.eecs.berkeley.edu/pubs/700.html>.
- Lee, E. A. and P. Varaiya, 2011: *Structure and Interpretation of Signals and Systems*. LeeVaraiya.org, 2nd ed. Available from: <http://LeeVaraiya.org>.
- Lee, E. A. and H. Zheng, 2005: Operational semantics of hybrid systems. In Morari, M. and L. Thiele, eds., *Hybrid Systems: Computation and Control (HSCC)*, Springer-Verlag, Zurich, Switzerland, vol. LNCS 3414, pp. 25–53. doi:10.1007/978-3-540-31954-2_2.
- , 2007: Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *EMSOFT*, ACM, Salzburg, Austria, pp. 114 – 123. doi:10.1145/1289927.1289949.
- Leung, M.-K., T. Mandl, E. A. Lee, E. Latronico, C. Shelton, S. Tripakis, and B. Lickly, 2009: Scalable semantic annotation using lattice-based ontologies. In *International Conference on Model Driven Engineering Languages and Systems (MODELS)*, ACM/IEEE, Denver, CO, USA. Available from: <http://chess.eecs.berkeley.edu/pubs/611.html>.

- Lickly, B., 2012: Static model analysis with lattice-based ontologies. PhD Thesis Technical Report No. UCB/EECS-2012-212, EECS Department, University of California, Berkeley. Available from: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-212.html>.
- Lickly, B., C. Shelton, E. Latronico, and E. A. Lee, 2011: A practical ontology framework for static model analysis. In *International Conference on Embedded Software (EMSOFT)*, ACM, pp. 23–32. Available from: <http://chess.eecs.berkeley.edu/pubs/862.html>.
- Lin, Y., R. Mullenix, M. Woh, S. Mahlke, T. Mudge, A. Reid, and K. Flautner, 2006: SPEX: A programming language for software defined radio. In *Software Defined Radio Technical Conference and Product Exposition*, Orlando. Available from: <http://www.eecs.umich.edu/~sdrp/publications.php>.
- Liskov, B. and S. Zilles, 1974: Programming with abstract data types. *ACM Sigplan Notices*, **9**(4), 50–59. doi:10.1145/942572.807045.
- Liu, J., B. Wu, X. Liu, and E. A. Lee, 1999: Interoperation of heterogeneous CAD tools in Ptolemy II. In *Symposium on Design, Test, and Microfabrication of MEMS/MOEMS*, Paris, France. Available from: <http://ptolemy.eecs.berkeley.edu/publications/papers/99/toolinteraction/>.
- Liu, X. and E. A. Lee, 2008: CPO semantics of timed interactive actor networks. *Theoretical Computer Science*, **409**(1), 110–125. doi:10.1016/j.tcs.2008.08.044.
- Liu, X., E. Matsikoudis, and E. A. Lee, 2006: Modeling timed concurrent systems. In *CONCUR 2006 - Concurrency Theory*, Springer, Bonn, Germany, vol. LNCS 4137, pp. 1–15. doi:10.1007/11817949_1.
- Lynch, N., R. Segala, F. Vaandrager, and H. Weinberg, 1996: Hybrid I/O automata. In Alur, R., T. Henzinger, and E. Sontag, eds., *Hybrid Systems III*, Springer-Verlag, vol. LNCS 1066, pp. 496–510.
- Lzaro Cuadrado, D., A. P. Ravn, and P. Koch, 2007: Automated distributed simulation in Ptolemy II. In *Parallel and Distributed Computing and Networks (PDCN)*, Acta Press.
- Maler, O., Z. Manna, and A. Pnueli, 1992: From timed to hybrid systems. In *Real-Time: Theory and Practice, REX Workshop*, Springer-Verlag, pp. 447–484.
- Malik, S., 1994: Analysis of cyclic combinational circuits. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, **13**(7), 950–956.
- Manna, Z. and A. Pnueli, 1992: *The Temporal Logic of Reactive and Concurrent Systems*. Springer, Berlin.
- , 1993: Verifying hybrid systems. In *Hybrid Systems*, vol. LNCS 736, pp. 4–35.
- Maraninchi, F. and T. Bhouhadiba, 2007: 42: Programmable models of computation for a component-based approach to heterogeneous embedded systems. In *6th ACM International Conference on Generative Programming and Component Engineering (GPCE)*, Salzburg, Austria, pp. 1–3.
- Maraninchi, F. and Y. Rémond, 2001: Argos: an automaton-based synchronous language. *Computer Languages*, (27), 61–92.

- Matic, S., I. Akkaya, M. Zimmer, J. C. Eidson, and E. A. Lee, 2011: Prides model on a distributed testbed emulating smart grid real-time applications. In *Innovative Smart Grid Technologies (ISGT-EUROPE)*, IEEE, Manchester, UK. Available from: <http://chess.eecs.berkeley.edu/pubs/857.html>.
- Matsikoudis, E., C. Stergiou, and E. A. Lee, 2013: On the schedulability of real-time discrete-event systems. In *International Conference on Embedded Software (EMSOFT)*, ACM, Montreal, Canada.
- Matthews, S. G., 1995: An extensional treatment of lazy data flow deadlock. *Theoretical Computer Science*, **151**(1), 195–205.
- Messerschmitt, D. G., 1984: A tool for structured functional simulation. *IEEE Journal on Selected Areas in Communications*, **SAC-2**(1).
- Mills, D. L., 2003: A brief history of NTP time: confessions of an internet timekeeper. *ACM Computer Communications Review*, **33**.
- Milner, R., 1978: A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, **17**, 348–375.
- , 1980: *A Calculus of Communicating Systems*, vol. 92 of *Lecture Notes in Computer Science*. Springer.
- Misra, J., 1986: Distributed discrete event simulation. *ACM Computing Surveys*, **18**(1), 39–65.
- Modelica Association, 2009: Modelica®- a unified object-oriented language for physical systems modeling: Language specification version 3.1. Report. Available from: <http://www.Modelica.org>.
- Moir, I. and A. Seabridge, 2008: *Aircraft Systems: Mechanical, Electrical, and Avionics Subsystems Integration*. AIAA Education Series, Wiley, third edition ed.
- Moreira, O., T. Basten, M. Geilen, and S. Stuijk, 2010: Buffer sizing for rate-optimal single-rate dataflow scheduling revisited. *IEEE Transactions on Computers*, **59**(2), 188–201. doi:10.1109/TC.2009.155.
- Morris, J. H. and P. Henderson, 1976: A lazy evaluator. In *Conference on the Principles of Programming Languages (POPL)*, ACM.
- Mosterman, P. J. and H. Vangheluwe, 2004: Computer automated multi-paradigm modeling: An introduction. *Simulation: Transactions of the Society for Modeling and Simulation International Journal of High Performance Computing Applications*, **80**(9), 433–450.
- Motika, C., H. Fuhrmann, and R. v. Hanxleden, 2010: Semantics and execution of domain specific models. In *Workshop Methodische Entwicklung von Modellierungswerkzeugen (MEMWe 2010) at conference INFORMATIK 2010*, Bonner Köllen Verlag, Leipzig, Germany, vol. GI-Edition – Lecture Notes in Informatics (LNI).
- Murata, T., 1989: Petri nets: Properties, analysis and applications. *Proceedings of IEEE*, **77**(4), 541–580. doi:10.1109/5.24143.

- Murthy, P. K. and S. S. Bhattacharyya, 2006: *Memory Management for Synthesis of DSP Software*. CRC Press.
- Murthy, P. K. and E. A. Lee, 2002: Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, **50**(8), 2064–2079. doi:10.1109/TSP.2002.800830.
- Object Management Group (OMG), 2007: A UML profile for MARTE, beta 1. OMG Adopted Specification ptc/07-08-04, OMG. Available from: <http://www.omg.org/omgmarte/>.
- , 2008a: System modeling language specification v1.1. Tech. rep., OMG. Available from: <http://www.sysmlforum.com>.
- , 2008b: A UML profile for MARTE, beta 2. OMG Adopted Specification ptc/08-06-09, OMG. Available from: <http://www.omg.org/omgmarte/>.
- Olson, A. G. and B. L. Evans, 2005: Deadlock detection for distributed process networks. In *ICASSP*.
- Parks, T. M., 1995: Bounded scheduling of process networks. Ph.D. Thesis Tech. Report UCB/ERL M95/105, UC Berkeley. Available from: <http://ptolemy.eecs.berkeley.edu/papers/95/parksThesis>.
- Parks, T. M. and D. Roberts, 2003: Distributed process networks in Java. In *International Parallel and Distributed Processing Symposium*, Nice, France.
- Patel, H. D. and S. K. Shukla, 2004: *SystemC Kernel Extensions for Heterogeneous System Modelling*. Kluwer.
- Pino, J. L., T. M. Parks, and E. A. Lee, 1994: Automatic code generation for heterogeneous multiprocessors. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Adelaide, Australia, pp. 445–448. doi:10.1109/ICASSP.1994.389626.
- Pree, W. and J. Templ, 2006: Modeling with the timing definition language (TDL). In *Automotive Software Workshop San Diego (ASWSD) on Model-Driven Development of Reliable Automotive Services*, Springer, San Diego, CA, LNCS.
- Press, W. H., S. Teukolsky, W. T. Vetterling, and B. P. Flannery, 1992: *Numerical Recipes in C: the Art of Scientific Computing*. Cambridge University Press.
- Prochnow, S. and R. von Hanxleden, 2007: Statechart development beyond WYSIWYG. In *International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, ACM/IEEE, Nashville, TN, USA.
- Ramadge, P. and W. Wonham, 1989: The control of discrete event systems. *Proceedings of the IEEE*, **77**(1), 81–98.
- Reed, G. M. and A. W. Roscoe, 1988: Metric spaces as models for real-time concurrency. In *3rd Workshop on Mathematical Foundations of Programming Language Semantics*, London, UK, pp. 331–343.
- Rehof, J. and T. A. Mogensen, 1996: Tractable constraints in finite semilattices. In *SAS '96: Proceedings of the Third International Symposium on Static Analysis*, Springer-

- Verlag, London, UK, ISBN 3-540-61739-6, pp. 285–300.
- Rehof, J. and T. . Mogensen, 1999: Tractable constraints in finite semilattices. *Science of Computer Programming*, **35**(2-3), 191–221.
- Ritchie, D. M. and K. L. Thompson, 1974: The UNIX time-sharing system. *Communications of the ACM*, **17**(7), 365 – 375.
- Rodiers, B. and B. Lickly, 2010: Width inference documentation. Technical Report UCB/EECS-2010-120, EECS Department, University of California. Available from: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-120.html>.
- Sander, I. and A. Jantsch, 2004: System modeling and transformational design refinement in ForSyDe. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, **23**(1), 17–32.
- Schruben, L. W., 1983: Simulation modeling with event graphs. *Communications of the ACM*, **26**(11), 957–963.
- , 1995: Building reusable simulators using hierarchical event graphs. In *Winter Simulation Conference (WSC 95)*, IEEE Computer Society, Los Alamitos, CA, USA, ISBN 0-7803-3018-8, pp. 472–475.
- Shapiro, F. R., 2006: *The Yale Book of Quotations*. Yale University Press.
- Sih, G. C. and E. A. Lee, 1993a: A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, **4**(2), 175–187. doi:10.1109/71.207593.
- , 1993b: Declustering : A new multiprocessor scheduling technique. *IEEE Transactions on Parallel and Distributed Systems*, **4**(6), 625–637. doi:10.1109/71.242160.
- Simitci, H., 2003: *Storage Network Performance Analytics*. Wiley.
- Smith, N. K., 1929: *Immanuel Kant's Critique of Pure Reason*. Macmillan and Co. Available from: <http://www.hkbu.edu.hk/~ppp/cpr/toc.html>.
- Som, T. K. and R. G. Sargent, 1989: A formal development of event graph models as an aid to structured and efficient simulation programs. *ORSA Journal on Computing*, **1**(2), 107–125.
- Spönemann, M., H. Fuhrmann, R. v. Hanxleden, and P. Mutzel, 2009: Port constraints in hierarchical layout of data flow diagrams. In *17th International Symposium on Graph Drawing (GD)*, Springer, Chicago, IL, USA, vol. LNCS. Available from: <http://rtsys.informatik.uni-kiel.de/~biblio/downloads/papers/gd09.pdf>.
- Srini, V., 1986: An architectural comparison of dataflow systems. *Computer*, **19**(3).
- Sriram, S. and S. S. Bhattacharyya, 2009: *Embedded Multiprocessors: Scheduling and Synchronization*. CRC press, 2nd ed.

- Stark, E. W., 1995: An algebra of dataflow networks. *Fundamenta Informaticae*, **22**(1-2), 167–185.
- Stephens, R., 1997: A survey of stream processing. *Acta Informatica*, **34**(7).
- Stuijk, S., M. C. Geilen, and T. Basten, 2008: Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Transactions on Computers*, **57**(10), 1331–1345. doi:10.1109/TC.2008.58.
- Thies, W., M. Karczmarek, and S. Amarasinghe, 2002: StreamIt: A language for streaming applications. In *11th International Conference on Compiler Construction*, Springer-Verlag, Grenoble, France, vol. LNCS 2304. doi:10.1007/3-540-45937-5_14.
- Thies, W., M. Karczmarek, J. Sermulins, R. Rabbah, and S. Amarasinghe, 2005: Teleport messaging for distributed stream programs. In *Principles and Practice of Parallel Programming (PPoPP)*, ACM, Chicago, USA. doi:10.1145/1065944.1065975.
- Tripakis, S., C. Stergiou, C. Shaver, and E. A. Lee, 2013: A modular formal semantics for Ptolemy. *Mathematical Structures in Computer Science*, **23**, 834–881. Available from: <http://chess.eecs.berkeley.edu/pubs/999.html>, doi:10.1017/S0960129512000278.
- Turjan, A., B. Kienhuis, and E. Deprettere, 2003: Solving out-of-order communication in Kahn process networks. *Journal on VLSI Signal Processing-Systems for Signal, Image, and Video Technology*, **40**, 7 – 18. doi:10.1007/s11265-005-4935-5.
- University of Pennsylvania MoBIES team, 2002: HSIF semantics (version 3, synchronous edition). Tech. Rep. Report, University of Pennsylvania.
- von der Beeck, M., 1994: A comparison of Statecharts variants. In Langmaack, H., W. P. de Roever, and J. Vytzil, eds., *Third International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, Springer-Verlag, Lübeck, Germany, vol. 863 of *Lecture Notes in Computer Science*, pp. 128–148.
- von Hanxleden, R., 2009: SyncCharts in C - A proposal for light-weight deterministic concurrency. In *ACM Embedded Software Conference (EMSOFT)*, pp. 11–16. doi:10.1145/1629335.1629366.
- Wiener, N., 1948: *Cybernetics: Or Control and Communication in the Animal and the Machine*. Librairie Hermann & Cie, Paris, and MIT Press, Cambridge, MA.
- Xiong, Y., 2002: An extensible type system for component-based design. Ph.D. Thesis Technical Memorandum UCB/ERL M02/13, University of California, Berkeley, CA 94720. Available from: <http://ptolemy.eecs.berkeley.edu/papers/02/typeSystem>.
- Yates, R. K., 1993: Networks of real-time processes. In Best, E., ed., *Proc. of the 4th Int. Conf. on Concurrency Theory (CONCUR)*, Springer-Verlag, vol. LNCS 715.
- Zeigler, B., 1976: *Theory of Modeling and Simulation*. Wiley Interscience, New York.
- Zeigler, B. P., H. Praehofer, and T. G. Kim, 2000: *Theory of Modeling and Simulation*. Academic Press, 2nd ed.

- Zhao, Y., 2009: On the design of concurrent, distributed real-time systems. Ph.D. Thesis Technical Report UCB/EECS-2009-117, EECS Department, UC Berkeley. Available from: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-117.html>.
- Zhao, Y., E. A. Lee, and J. Liu, 2007: A programming model for time-synchronized distributed real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, IEEE, Bellevue, WA, USA, pp. 259 – 268. doi:10.1109/RTAS.2007.5.
- Zou, J., 2011: From ptides to ptidyos, designing distributed real-time embedded systems. PhD Dissertation Technical Report UCB/EECS-2011-53, UC Berkeley. Available from: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-53.html>.
- Zou, J., J. Auerbach, D. F. Bacon, and E. A. Lee, 2009a: PTIDES on flexible task graph: Real-time embedded system building from theory to practice. In *Languages, Compilers, and Tools for Embedded Systems (LCTES)*, ACM, Dublin, Ireland. Available from: <http://chess.eecs.berkeley.edu/pubs/531.html>.
- Zou, J., S. Matic, E. A. Lee, T. H. Feng, and P. Derler, 2009b: Execution strategies for Ptides, a programming model for distributed embedded systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, IEEE, San Francisco, CA. Available from: <http://chess.eecs.berkeley.edu/pubs/529.html>.

信息物理融合系统 (CPS) 设计、建模与仿真

基于Ptolemy II平台 System Design, Modeling, and Simulation Using Ptolemy II

本书是一本系统论述CPS (集成了计算、网络和物理过程的信息物理融合系统) 建模问题的专著, 不仅由浅入深地逐一剖析CPS的建模问题, 而且还详细介绍了Ptolemy II在应对CPS模型问题时的各种解决方案, 并结合Ptolemy II系统平台 (集系统设计、建模与仿真于一体) 介绍了大量分层、异构系统的模型实例。本书的目标是让读者理解现代建模技术在设计过程中所能发挥的巨大作用, 并掌握使用这些技术的方法。书中介绍的技术可以用于诸如嵌入式软件、机械、电气、控制、生物信息等领域的系统建模, 尤其适合于那些由不同领域设计元素组合的异构混合系统的建模。另外, 书中提到的所有方法和实例都可以在项目网站 (<http://ptolemy.org/systems>) 上下载开源的代码。

本书适合作为高等院校相关专业“嵌入式系统”课程的教材或教学参考书, 也可作为专业技术人员在CPS系统建模过程中的参考书。

作者简介

爱德华·阿什福德·李 (Edward Ashford Lee) 曾任加州大学伯克利分校电子工程与计算机科学系主任, 现为该系Robert S. Pepper特聘教授。主要研究方向是嵌入式与实时计算系统的设计、建模和模拟。他是嵌入式系统领域的著名学者, 也是CPS研究的倡导者和引领者之一。他领导的团队成功开发了Ptolemy项目, 研发了Ptolemy Classic和Ptolemy II系统, 这是一个非常优秀的开源嵌入式系统研究与开发平台。他还是加州大学伯克利分校CHESS (混合及嵌入式软件和系统中心) 创始主任。



Lee教授拥有加州大学伯克利分校博士学位, 麻省理工学院理学硕士学位, 以及耶鲁大学学士学位。他合著了5本著作, 其中包括《嵌入式系统: CPS方法》。他是IEEE Fellow, 于1997年获得工程教育领域的Frederick Emmons Terman奖, 1987年获得美国国家科学基金会的青年研究者总统奖。

投稿热线: (010) 88379604
客服热线: (010) 88378991 88361066
购书热线: (010) 68326294 88379649 68995259

华章网站: www.hzbook.com
网上购书: www.china-pub.com
数字阅读: www.hzmedia.com.cn



封面设计: 金易 林

上架指导: 计算机/嵌入式系统

ISBN 978-7-111-55843-9



9 787111 558439 >

定价: 79.00元